

NASA/CR-2013-217961



Investigating System Dependability Modeling Using AADL

*Brendan Hall, Kevin R. Driscoll, and Gabor Madl
Honeywell International, Inc., Golden Valley, Minnesota*

February 2013

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- **TECHNICAL PUBLICATION.** Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- **TECHNICAL MEMORANDUM.** Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- **CONTRACTOR REPORT.** Scientific and technical findings by NASA-sponsored contractors and grantees.

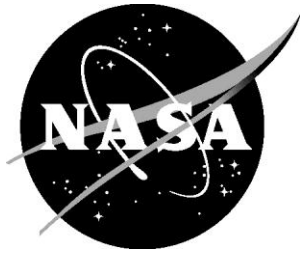
- **CONFERENCE PUBLICATION.** Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- **SPECIAL PUBLICATION.** Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- **TECHNICAL TRANSLATION.** English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at <http://www.sti.nasa.gov>
- E-mail your question to help@sti.nasa.gov
- Fax your question to the NASA STI Information Desk at 443-757-5803
- Phone the NASA STI Information Desk at 443-757-5802
- Write to:
STI Information Desk
NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320

NASA/CR–2013-217961



Investigating System Dependability Modeling Using AADL

*Brendan Hall, Kevin R. Driscoll, and Gabor Madl
Honeywell International, Inc., Golden Valley, Minnesota*

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL10AB32T

February 2013

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

This report describes Architecture Analysis & Design Language (AADL) models for a diverse set of fault-tolerant, embedded data networks and describes the methods and tools used to create these models. It also includes error models per the AADL Error Annex. Some networks were modeled using Error Detection Isolation Containment Types (EDICT). This report gives a brief description for each of the networks, a description of its modeling, the model itself, and evaluations of the tools used for creating the models. The methodology includes a naming convention that supports a systematic way to enumerate all of the potential failure modes.

Contents

1	Introduction	4
1.1	Scope	4
1.2	Motivation and Modeling Intent	4
1.3	Tools	5
2	Background	6
2.1	AADL Modeling Overview	6
2.1.1	What AADL Can Do	6
2.1.2	AADL Language Basics	6
2.1.3	Three Levels of Specification	7
2.2	Overview of the AADL Error Annex	7
2.2.1	Model of Computation	8
2.2.2	Guard Behavior	8
2.3	Overview of the EDICT Error Modeling Approach	9
2.3.1	Error Propagation in EDICT	9
2.3.2	Error Mitigators in EDICT	9
2.3.3	Running Analyzers	10
3	Case Study Selection	11
3.1	Network-Centric Modeling	11
3.2	Selected Protocols	11
4	AADL Modeling of Fault-Tolerant Systems	13
4.1	Modeling Approach and Naming Conventions	13
5	Case Study: SAFEbus	15
5.1	SAFEbus Protocol Description	15
5.2	Modeling SAFEbus Using AADL	17
5.2.1	Bus Access vs. Data Connections for Buses	17
5.3	Modeling Error Propagation in SAFEbus Using the AADL Error Annex	19
5.3.1	Host Error Model	20
5.3.2	BridgeComparator Error Model	20
5.3.3	BusInterfaceUnit Error Model	21
5.3.4	BackplaneTransceiverLogic Error Model	21
5.3.5	BusDevice Error Model	21
5.3.6	Error Mitigation Modeling - Sender Side	21
5.3.7	Error Mitigation Modeling - Receiver Side	22
6	Case Study: BRAIN	23
6.1	Protocol Overview	23
6.2	Modeling BRAIN Using AADL	24
6.2.1	Bus Access Connections	25
6.3	Modeling Error Propagation in BRAIN Using the AADL Error Annex	25
6.3.1	Limitations of the Initial BRAIN Model	27

7	Case Study: SPIDER	29
7.1	SPIDER/ROBUS Protocol Description	29
7.2	Modeling SPIDER Using AADL	29
7.3	Modeling Error Propagation in SPIDER Using the AADL Error Annex	31
7.3.1	Processing Element Error Model	31
7.3.2	BusInterfaceUnit Error Model	32
7.3.3	RedundancyManagementUnit Error Model	33
7.3.4	Error Mitigation Modeling - Sender Side	33
7.3.5	Error Mitigation Modeling - Receiver Side	33
8	Case Study: TTP	34
8.1	TTP Protocol Description	34
8.2	Modeling TTP Using AADL	35
8.2.1	Modeling Buses	38
8.2.2	TTP Hub Model	39
8.3	Modeling Error Propagation in TTP Using the AADL Error Annex	39
8.3.1	Driver Modeling	40
8.3.2	Channel Modeling	41
8.3.3	TTP_Hub Modeling	41
9	Findings and Discussion	42
9.1	Benefits and Overheads of the Systematic Fault Taxonomy	42
9.2	Role of Multiple Layers of Abstraction	42
9.3	Completeness of Modeling and Analysis	43
9.4	Obtaining Probabilities	43
9.5	Composition of Heterogeneous Models of Computation	43
9.6	Experiments Using Integrated Behavior and Probabilistic Models	44
10	Concluding Remarks	45
11	Acronyms and Initialisms	46
	References	47
A	Real-time Availability Integrity Language (RAIL)	48
A.1	Introduction	48
A.2	Background	48
A.2.1	Discrete Event Systems	49
A.2.2	Petri-nets	49
A.2.3	Applying Petri-nets for the Modeling of Ring Networks	50
A.2.4	Finite State Machines	51
A.3	Modeling Fault-Tolerant Communication in Distributed Systems	51
A.3.1	Applying RAIL for the Analysis of Braided Ring Topologies	51
A.3.2	RAIL Execution Semantics	51
A.3.3	Comparison with Petri-nets	54
A.3.4	Reconstitution	55
A.4	Automated Verification of RAIL Models	56
A.4.1	Formal Modeling of RAIL in SAL	56
A.5	Conclusion	57

1 Introduction

The documented work was performed under NASA Task Order NNL10AB32T, Validation and Verification of Safety-Critical Integrated Distributed Systems—Area 2.

1.1 Scope

This document is intended to satisfy the requirements for Deliverable 5.1.6 under Task 4.1.2.1 of this Task Order. The aim of this work is to evaluate current capability and expressiveness of Architecture Analysis & Design Language (AADL) to capture the behavior of real-world fault-tolerant systems. By sharing the case-studies herein, we hope to support the evolution of the AADL standard. The work in this document is mostly focused on the activities of Year 1. Note that as a result of tooling available during this period, this work used Version 1 of the AADL Error Annex. Many initial syntactic findings have already been addressed as Version 2 of this annex was drafted.

This document accompanies Deliverable 5.1.7, which comprises the AADL models in electronic form. These can be downloaded from the NASA DASHlink site (AFCS-Distributed Systems). [<https://c3.nasa.gov/dashlink/projects/79/>]. However, it is emphasized that these models reflect ongoing work under Task Order NNL10AB32T. Given the length of the research program, the electronic models are expected to be continuously revised and updated as further progress is made.

1.2 Motivation and Modeling Intent

One obstacle to the broad adoption of formal methods is the gap between the tools in current use by practicing engineers and the tools that support formal methods. The additional labor required to bridge this gap creates a disincentive for designers. In addition, the creation of the manual abstractions that are often required to implement a formal model of a real-world system is an area of significant risk. This risk is largely due to the different experience bases associated with the systems engineering professionals developing the real-world system and the experience base of formal method practitioners. A practicing systems engineer can rarely afford the luxury of spending the significant effort required to master a formal notation. In addition, “tribal knowledge” associated with many real-world domains has rarely been sufficiently captured into a suitable formal notation that non-domain experts can understand. This lack of notation results in potential risk, as the abstractions used within formal system models miss important details and assumptions about the system behavior and environment; for example, the assumed failure modes of system components. For distributed systems this is especially important, since there are often unstated assumptions about distributed data congruency and the required degree of replica determinism.

This situation is also compounded by the limited traceability often encountered with formal analysis and the associated tooling, especially model checkers. Often these formal models may fail to scale to represent real-world system size.¹ Hence, they may be considered academic and non-relevant. In the other direction, researchers and developers of formal method tools would like to use “real-world examples” to test research ideas and tool development; however, creating such examples only for testing is prohibitively expensive, and gaining access to the tribal knowledge almost impossible. It is therefore desirable to develop technology and languages that bridge this gap between real-world system model development and formal systems analysis. In recent years,

¹Consider the formal analysis of TT Ethernet, where the formal model initially scaled to a dual-channel system with five end-systems, whereas practical systems could comprise more than 9 switches and in excess of 30 end-systems.

AADL[1] has gained increasing popularity within both the research and industrial communities. This increase in popularity has been aided by the following attributes:

- The language openness and the standardization of the core language and its annexes.
- The vibrant AADL research community that constantly strive to drive increasing levels of formalism into the language semantics.
- The proliferation of research tools that are growing around the emerging standard to support systems analysis.
- The AADL language is inherently extensible. The application of custom property sets and/or custom annexes can target the core language mechanisms to cover a wide range of modeling domains.

Given this cross-domain adoption and documented successes in supporting formal analysis of domain-driven models, AADL appears to be a good candidate as a cross-domain bridging technology.

Our primary focus is on the validation and verification of distributed systems and their associated dependency properties. For system dependability modeling, the AADL Error Annex [2] is of key interest as it supports modeling system dependability mechanisms. Since the original publication of this annex in 2005, several studies have attempted the application of AADL to system dependability modeling. To date, these studies have demonstrated promising levels of success. Joshi [3] presented a proof-of-concept strategy for the automated generation of system fault trees from suitably annotated AADL models. Rugina [4] demonstrated a more elaborate dependability analysis framework introducing a bridge between the AADL model and a back-end GSPN (Generalized Stochastic Petri Net) representation that can be processed by current dependability analysis tools. Hecht [5] extended this work and has also demonstrated the proof of concept generation of automated FMEAs from AADL models. Given the aforementioned successes, the promise of model-driven safety engineering appears to be on the horizon. The ability of AADL to capture different aspects of the system through dedicated annexes is a great step forward. As these annexes mature, we hope they will facilitate an integrated model of the system to be formally captured. This model may then serve as a central repository from which validation and verification activities can be driven. The emerging AADL Requirements Annex is also establishing the required mechanisms to introduce formal traceability among the model components.

We intend to apply and assess the capability of AADL to capture the critical attributes of real-world fault-tolerant distributed systems with related protocols. By doing this work, we hope use our lessons learned to provide feedback to aid the AADL language evolution. To this end, we have presented our observations and case studies to the AADL AS-2C working committee. We would like to acknowledge Peter Feiler of the SEI for his excellent mentorship and feedback related to our endeavors.

1.3 Tools

The AADL Error Annex work described in this report is based on AADL v1, Open Source AADL Tool Environment (OSATE) v1.5.8, and Error Annex plug-in version 1.1.7. All are freely available at <http://www.aadl.info>. The AADL model figures were created with the EDICT tool suite, available at <http://www.wvtechnology.com>. EDICT is based on the AADL v2 language.

2 Background

2.1 AADL Modeling Overview

AADL is an international standard (SAE 5506A) for predictable, model-based engineering of real-time and embedded computer systems. AADL was originally developed at Honeywell as the Meta-H tool, then later as the Avionics Architecture Description Language.

AADL development was funded primarily by DARPA and the U.S. Army. Bruce Lewis (U.S. Army) is the chair of the AADL subcommittee, and Peter Feiler (SEI, CMU) is the AADL technical lead. AADL is supported by the Open-Source AADL Tool Environment (OSATE), the EDICT tool suite, and TOPCASED, among other tools.

Intended fields of use for AADL include automotive, avionics, space, medical devices, and industrial control. Current users include Rockwell-Collins, General Dynamics, Airbus, European Space Agency, and Honeywell, among others.

2.1.1 What AADL Can Do

AADL functionality includes:

- Representing embedded systems as component-based architecture.
- Modeling component interaction as flows, service calls, and shared access.
- Modeling task execution and communication with precise timing semantics.
- Modeling execution platform and specifying application binding.
- Representing operational modes and fault-tolerant configurations.
- Supporting component evolution and large-scale development.
- Accommodating analysis, such as reliability and safety criticality through extensions.

2.1.2 AADL Language Basics

AADL has standardized both a graphical and textual syntax. The left side of Figure 1 demonstrates the graphical syntax for AADL language elements. In this report, we rely on the textual syntax to capture representative fault-tolerant systems.

The key AADL modeling categories [1] are as follows:

- *Data*: specifies the types of data exchanged between components.
- *Device*: represents a platform component such as a hardware unit.
- *Memory*: represents a platform storage unit.
- *Bus*: represents a platform component that can exchange data and control between other platform components.
- *Thread*: represents a unit of sequential execution, typically a software thread.
- *Process*: represents virtual address space, must contain thread(s).
- *Processor*: represents a platform component that schedules and executes threads/processes.
- *System*: represents a composite actor that may contain other components.
- *Subprogram*: represents source code function call.
- *Thread group*: logically groups threads within processes.

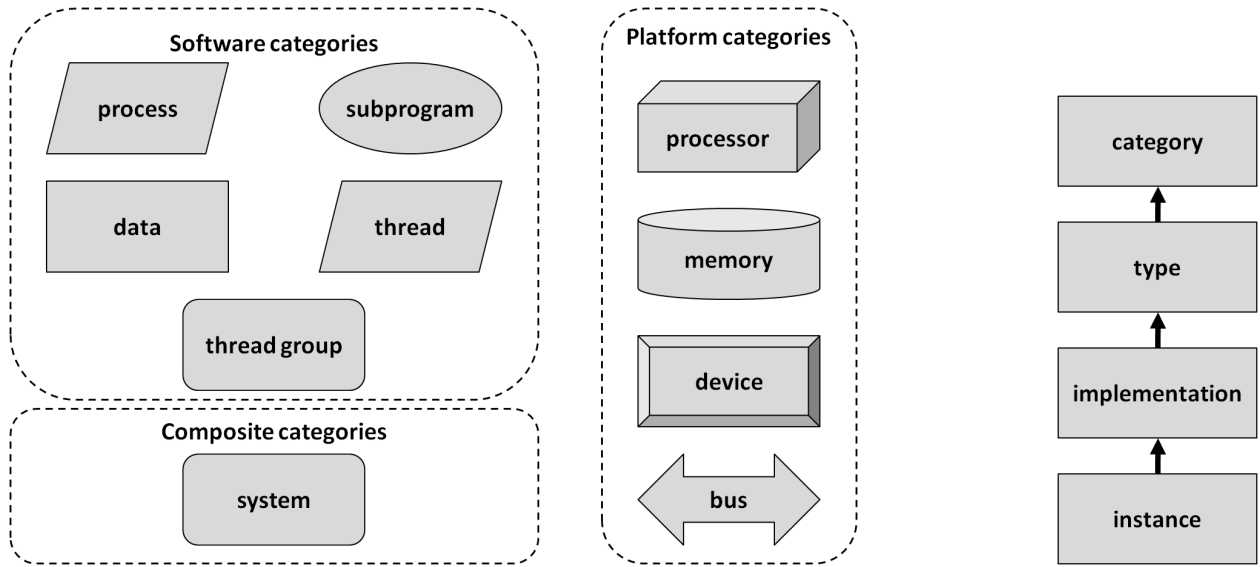


Figure 1. Left: AADL Graphical Syntax; Right: AADL Three Levels of Specification

2.1.3 Three Levels of Specification

AADL utilizes multiple levels of specification, as shown on the right side of Figure 1, starting with categories as described in Section 2.1.2. A type-level specification is derived from the categories, similar to the concept of classes used in software engineering. Types define external interfaces, ports, and a high-level view for data flows.

The second level of specification is implementation, which describes the subcomponents for the types, specifies internal connections between subcomponents, etc.

Finally, the instance level describes one instance of the implementation. Thus, AADL provides a way to create many instances of the same component, reducing the overall modeling effort.

2.2 Overview of the AADL Error Annex

The AADL Error Annex extends AADL with error modeling concepts, including sourcing errors, propagation, and mitigation. The AADL Error Annex is standardized as SAE Aerospace Standard 5506/1[2].

The OSATE tool includes an Error Annex plug-in. This plug-in can parse Error Annex specifications and can perform syntax checking on the specification. The Error Annex plug-in can then translate the specification into Extensible Markup Language (XML) format, to be used by other tools. An example of such back-end analysis is the automated generation of system fault trees as demonstrated by Joshi et al.[3]

The AADL Error Annex is a relatively new development, with activity and complementary approaches available for error modeling, such as the concepts implemented in EDICT.

2.2.1 Model of Computation

The Model of Computation (MoC) for the AADL Error Annex is essentially a network of finite state machines, extended with stochastic probabilities. $M = \{S, s_0, E, T, \Sigma, \gamma, \delta\}$:

- S is the set of states.
- $s_0 \in S$ is initial state.
- E is the set of events.
- $T : S \times E \times S$ is the set of transitions.
- Σ is a finite alphabet of symbols, specifying probabilities or rates.
- The mapping γ is the guard function.
- The labeling function $\delta : E \rightarrow \Sigma$ specifies probabilities or fixed rates for events.

Events can be further categorized into: *(i)* error events that can be sourced by the component with a given probability and that play roles in triggering internal transitions; *(ii)* error propagations express events passing between different automata and can be further divided into **in** and **out** error propagations; *(iii)* repair events.

The AADL Error Annex has no explicit notion of time, although a partial ordering of error events and propagations is implied by the transitions.

2.2.2 Guard Behavior

Guards express mappings between different error events and from states to events. Guards can be useful in modeling how a particular error manifestation can lead to different errors as it propagates through the system. Guards can also “mask” events—prevent certain error propagations from leaving or reaching another error model, e.g., via voting. Guards can be further divided into **Guard_In**, **Guard_Out**, **Guard_Event**, and **Guard_Transition**.

Out guards will take either an **in** error propagation event or an error state and translate the event or state to an **out** error propagation event. **Out** guards cannot refer to error events sourced in the same component, and they pass error propagation events through without any associated state change.

In guards map either an **in** error propagation event or an error state to an error event. The error event mapped by the **in** guard can then be used in internal transitions of the automata. Neither **in** nor **out** guards can be cascaded together.

Guard_Events are raised whenever a component receives a specific error propagation. Finally, **Guard_Transitions** may be triggered by **Guard_Events**.

The AADL Error Annex defines guard modes that extend the guard functionality with a finite state machine model. Thus, a guard can be in multiple modes and can switch guard modes depending on the error propagation and error events it receives or sources.

2.3 Overview of the EDICT Error Modeling Approach

EDICT is a tool suite developed by WW Tech for model-based design of dependable systems based on Eclipse [<http://www.eclipse.org/>]. It builds on OSATE compilers to process AADL model specifications and can import AADL files and visualize designs on its Graphical User Interface (GUI).

Error semantics are the basic units of error propagation in EDICT. Currently, EDICT specifies the following error semantics by default:

- *BA-Crash*: representing benign asymmetric crash condition.
- *BA-Omission*: representing benign asymmetric omission error condition.
- *BS-FailStop*: representing benign symmetric failstop condition.
- *TA-Early*: representing timing asymmetric early error condition.
- *TA-Fast*: representing timing asymmetric fast error condition.
- *TA-Late*: representing timing asymmetric late error condition.
- *TA-Slow*: representing timing asymmetric slow error condition.
- *TS-Early*: representing timing symmetric early error condition.
- *TS-Fast*: representing timing symmetric fast error condition.
- *TS-Late*: representing timing symmetric late error condition.
- *TS-Slow*: representing timing symmetric slow error condition.
- *VA-Arbitrary*: representing value asymmetric arbitrary error condition.
- *VA-Range*: representing value asymmetric range error condition.
- *VS-Arbitrary*: representing value symmetric arbitrary error condition.
- *VS-BitError*: representing value symmetric bit-error condition.
- *VS-Range*: representing value symmetric range error condition.

Users can also specify their own error semantics using EDICT. In the next step, *Component Error Semantics* are defined to assign persistence (permanent or temporary) and occurrence (probability) to the error to be used with component types. Component error semantics are then assigned to specific component instances in *Component Error Models*.

Component error models specify the types of errors exhibited by the component (as specified in component error semantics) as well as transformed errors. Transformed errors are a similar concept to AADL Error Annex Out guards; they specify that certain error semantics should be transformed to another to express different error manifestations along the error propagation.

2.3.1 Error Propagation in EDICT

Errors propagate in EDICT as specified by the component error models and error mitigators. EDICT can auto-generate component error models for components with no particular associated error models. In the implementation of EDICT used for this study, the error propagation is stateless; component error models have no associated states. Moreover, there is no notion of time for error propagation.

2.3.2 Error Mitigators in EDICT

The error mitigators in EDICT include:

- Mask certain error semantics to express that the component mitigates that type of error.
- Mask certain outputs of a component to express directional error propagation.
- Transform one particular type of error semantics to another.
- Express detection of certain types of errors.

2.3.3 Running Analyzers

Once the component error models and mitigators are in place, users can run error analyzers. The user must specify component sourcing for the error. EDICT will then run the error propagation and mitigation analysis and display results on its GUI. EDICT's analyzers and reporting functionality are under development and likely to include future improvements such as the specification of error cross influences.

3 Case Study Selection

This section discusses our rationale for the selection of the case study network and protocol technologies.

3.1 Network-Centric Modeling

A key area of focus for the Assurance of Flight Critical Systems (AFCS) Subproject element is distributed systems. In such systems, the quality of the communication between the distributed system components serves as the system foundation. Consequently, it may be argued that network technologies and associated communication services comprise the most important aspects of the fault-tolerant system. Conceptually, the network technology is the “glue” for system components. With good glue, a dependable system can be made of unreliable components because it enables components to be replicated and composed into configurations that support higher degrees of availability or integrity. However, without this glue, a dependable system cannot be developed, regardless of the quality of the components.

With respect to a system’s dependability, the communication system policies usually make a good proxy for general system-level behavior. In addition, the data network may provide a number of services to aid system fault-tolerant replication—time synchronization, distributed agreement, and consensus, for example. The quality of each service must match application requirements.

During the first year of research, our AADL modeling focused on the data network’s communication services. This territory is not well-explored within AADL. Often, the details of this layer are abstracted out of AADL models that have traditionally focused on software behavior. In such models, network based connectivity is usually relegated to architecturally passive bus component abstractions. Given the importance of this layer to the system’s fault tolerance discussed above, this is an interesting dichotomy.

3.2 Selected Protocols

To explore network behavior, we selected a set of network architectures and technologies that demonstrate different fault-containment strategies.

The first protocol is SAFEbus. Leveraging self-checking paired configurations for its major components (host-processors, bus interface unit (BIU), and buses), the architecture provides a high degree of fault masking by comparing the data sourced from each half of a pair to be bit-for-bit identical. This method provides a high degree of fault coverage and is single Byzantine fault-tolerant. Using a layered protocol and a hierarchical Byzantine agreement strategy, it is possible for SAFEbus to deliver a high degree of application data consistency with relatively small bandwidth temporal overhead. The SAFEbus topology is a simple quad bus.

The next protocol is the Braided Ring Availability Integrity Network (BRAIN) architecture. This implements a ring topology and a “brother’s keeper” fault-tolerance philosophy. In this protocol, adjacent neighbors form self-checking pairs to support data relay (messages are checked for integrity at every hop as they traverses around the ring). The serialized segmented medium of the BRAIN requires a little more protocol than SAFEbus; however, this partitioned medium supports the potential for increased levels of fault-tolerance. Similar to SAFEbus, the BRAIN also enables hierarchical Byzantine agreement strategies to be deployed.

To consider non-self-checking protocols, we explore two additional protocols. The SPIDER/ROBUS has a point-to-point topology that is a bipartite graph consisting of a BIU fully interconnected with a redundancy management unit (RMU). This contrasts with SAFEbus and BRAIN in that it has no

shared media (no medium access control (MAC) protocol is required). While the lack of a MAC is a simplification, SPIDER/ROBUS needs additional protocols to handle diagnosis for fault tolerance. To this end, the SPIDER protocol is based on a Byzantine fault-tolerant broadcast strategy that leverages data path replication and fault-tolerant mid-value voting strategies to provide interactive data consistency.

Finally, to cover the lower end of the cost spectrum, we selected the Time-triggered Protocol (TTP) protocol in both hub and bus topologies. TTP is also interesting in that it has strong emergent protocol properties that manifest under certain failure conditions, specifically Byzantine failure, slightly out of specification (SoS) faults. Capturing these protocol dependability considerations within a complete abstraction framework that will allow them to be evaluated and analyzed with respect to the needs and requirements of the integrated total system dependability is one of our principal goals. In addition, investigating the bus and hub topologies of TTP allows us to consider different failure propagation patterns within a common framework. TTP also presents some interesting logical semantic vulnerabilities with respect to protocol violations.

In Year 2, we have expanded the architecture modeling to look at asynchronous protocols and higher-level application-driven fault-tolerant strategies.

4 AADL Modeling of Fault-Tolerant Systems

4.1 Modeling Approach and Naming Conventions

One goal of the AADL modeling framework is to make the modeling of system dependability more systematic and less prone to variation due to a particular modeler’s background, expertise, or experience base. Observing lessons learned from the simple modeling performed to date, it is apparent that such expertise differences can greatly influence the fidelity of the modeling assumptions.

In furtherance of this goal, we have developed a naming convention for the error events, propagations, and states used in the AADL Error Annex. The utility of such a naming convention became apparent only after working with several models by different modelers. The naming convention can function as a checklist to remind the modeler of all the possible errors and propagations that should be considered. The naming convention was adjusted several times as we gained experience.

One of the lessons learned involved the issue of errors propagating through intervening devices and/or layers of protocol that do not, and cannot, have any understanding of the semantics of that error. For example, to a stateless bus driver, “bits is bits.” It cannot know that the bits it is transferring mean “halt and catch fire,” and thus it cannot take any specific semantic-based mitigation actions. From this, we developed the concept of a naming convention in which we distinguish between faults that can be understood and correctly mitigated locally and those that have no local meaning and just “pass through.” A further augmentation to this naming convention may be to include the concept of error classes in which a local device may understand some semantics for a class of errors but may not understand all the semantics for the individual members of that class.

Where detailed protocol behavior underpins the dependability modeling assumptions, it is easy to overlook contributions of unforeseen protocol interaction and/or higher-level software/protocol interaction. For example, if a protocol does not implement software fault-containment strategies that are consistent with its fault-tolerance guarantees, these assumptions may fail. Consider a protocol that assumes data is identically replicated on redundant channels. The TTP bus is such a network. To support design flexibility TTP is configurable and incorporates modes that may delegate the responsibility of such channel data replication to software. Under such a modeling scheme, failure of the software layer may impact the protocol guarantees and lead to a disjoint system-level assumption. Similarly, the protocol may depend on correct software interaction, such as the required strobing of a life-sign during startup. Such vulnerabilities and dependencies therefore need to be captured to allow for rigorous system-level examination. Including the idea of a pass-through semantic tunnel in the naming convention helps the modeler by adding to its inherent checklist.

We hope that, through the AADL modeling work, we can develop a methodology to capture such interactions and dependencies, as a complete methodology has not yet been fully refined. Work performed to date suggests that a systematic, layered, formal naming and classification notation may be a useful first step toward achieving this long-term goal. The work we performed with EDICT also illustrated the power of applying systematic fault models to the modeling of software architectural failure contributions.

The naming system is shown here using the POSIX extended regular expression representation:

```
(([es]_?[ip]?)|p)_([bo]|([tv] [as] [dnq]))(_m_[^_]+)?(_nc)?((_remission|_repair)?_rate)?
```

The first character (*e*, *p*, or *s*) differentiates among error event, error propagation, and error state. These are the three main declared items within a fault model per the AADL Error Annex.

e $\Rightarrow$ error event
p $\Rightarrow$ error propagation
s $\Rightarrow$ error state

The next set of characters, from the first underscore up to the second underscore, denote the error manifestation. These are outlined below:

a $\Rightarrow$ error will manifest *Asymmetrically* among receivers, i.e., Byzantine error
b $\Rightarrow$ semantic-free *Babble* that leads to denial of service
d $\Rightarrow$ error is *Detectable* by inline acceptance testing
i $\Rightarrow$ error will manifest *Intermittently*, represent transient error behavior
n $\Rightarrow$ error is *Not* detectable by inline acceptance test
o $\Rightarrow$ *Omission*, fail-stop
p $\Rightarrow$ error will manifest *Persistently*, i.e., it is a permanent behavior
q $\Rightarrow$ data that has been flagged as *Questionable*, untrusted
s $\Rightarrow$ error will manifest *Symmetrically*, presenting the same value to all receivers
t $\Rightarrow$ a data *Temporal* error
v $\Rightarrow$ a data *Value* error

Some of these error manifestations must be interpreted in context. For example, consider the notion of a babbling fault, which denotes a semantic-free continuous disruption. At the driver level, this fault could be a simple shorted or stuck driver that prevents other member systems from utilizing a shared bus. In a system with replicated buses, such an error may be masked; however, at the protocol level, one babbling device may influence multiple (all) buses. The babbling from a protocol component may disrupt the entire system. Although some of these properties may be discoverable by error propagation analysis, at the time of writing, inclusion of the source as part of the error classifier has helped simplify the associated discussions.

The notion of semantic-free babbles is insufficient to capture the potential error propagation related to protocol semantic coupling. For this reason, an additional classifier *m* is appended to the error classification to indicate errors that may exhibit higher-level semantic *meaning* or coupling. By its very nature, such coupling is largely protocol specific, and as such it is envisaged that additional classification may follow *m* to differentiate different mechanism of protocol coupling. For example, in TTP, a cold-start *m_coldstart* frame can be differentiated with respect to an erroneous data frame *m_nframe*.

The next two optional items deal with errors that are meaningful only when including some context outside of the local component. These are:

m $\Rightarrow$ *Meaning* of error is outside of the local context
_nc $\Rightarrow$ data is *Not Consistent* with another copy/flow

The syntax for the *Meaning* error is the string *_m_* immediately followed by the name of the component that caused this error.

The last three items are used only for setting the probabilities or rates of events:

_rate $\Rightarrow$ error event rate
_remission_rate $\Rightarrow$ self-healing rate (for intermittent and transient errors)
_repair_rate $\Rightarrow$ repair rate

5 Case Study: SAFEbus

5.1 SAFEbus Protocol Description

Honeywell designed SAFEbus as a backplane for the Aircraft Information Management System (AIMS), which is the integrated modular avionics (IMA) for the Boeing B-777 airplane. SAFEbus is the only backplane or local area data network to become a standard (ARINC 659) that provides fail-op/fail-safe fault tolerance with near unity coverage for all of its components - signal lines, terminations, interface electronics, clock sources, and power supplies. This coverage includes tolerating a Byzantine fault. SAFEbus provides a time-based protocol that delivers messages with a precision on the order of 100 nanoseconds over a backplane network.

The SAFEbus interface logic consists of a BIU, clock, table memory, intermodule memory (IMM), and backplane transceivers. This logic is paired to provide immediate fault detection and containment, including providing a Byzantine fault barrier. The backplane bus lines are a unique form of dual-dual redundancy that provide high integrity and availability simultaneously.

SAFEbus consists of two self-checking buses (SCBs), A and B. Each SCB is itself composed of two buses, x and y. The interface logic, including the BIUs, is also duplicated. One of the BIUs transmits data on one of the buses within an SCB, and its partner transmits on the other bus. The data on any two buses from different BIUs are compared at the receiver. Only data that is bit-for-bit identical (x versus y) are written into the intermodule memories. Having four buses allows single-bus errors to be corrected on-the-fly and all double-bus errors to be detected. The receiving circuitry in the transmitting line replaceable modules (LRMs) also checks what is actually put on the bus. Such self-checking ensures a babbling LRM will be detected and will remove itself from SAFEbus. This removal is enforced by having each BIU control the other BIU drivers. If either BIU thinks it should not be transmitting, neither BIU can transmit.

SAFEbus and its self-checking approach provides near-perfect coverage. The checking at the receiving end provides near-perfect error detection coverage for many faults, including Byzantine faults [6]. It provides better coverage than signature-based error detection techniques (such as CRCs) [7] without simultaneously incurring the overhead of these schemes.

SAFEbus has a unique way of tolerating Byzantine faults. Because the transfer of a message from one LRM to another uses four fault zones, it is possible for it to tolerate one Byzantine fault. The Backplane Transceiver Logic (BTL) receivers are cross-linked to the two BIUs such that each receiving BIU gets a copy of the message from all four buses. This setup can be seen as the first round of the classical Byzantine exchange. Each BIU creates two four-bit status vectors, collectively called the “syndrome,” for each 16 bits received within a message. The first vector has a bit for each bus that identifies whether anything came in from that bus. The second vector is the result of the comparisons: $A_x = A_y$, $B_x = B_y$, $A_x = B_y$, $A_y = B_x$. The BIUs exchange their syndromes. From these eight bits, the two BIUs can determine which (if any) of the data bus inputs have arrived error-free. If an error-free source exists, both BIUs select it as the source data. This selection is the second round of the classical Byzantine exchange. It prevents Byzantine failures arriving from outside a pair from confusing a pair into thinking that one of the halves of the pair is faulty. While the syndrome exchange prevents a Byzantine fault from splitting a pair, an additional mechanism is needed for Byzantine agreement among pairs.

SAFEbus introduced a new method: hierarchical Byzantine agreement. In this method, a lower-level agreement prevents Byzantine faults from affecting a pair, as described above. An upper-level agreement only needs to send one bit of information from every receiving LRM of the message. This method is more efficient than previous methods that required a full exchange of all message content and/or elaborate use of signature schemes. The syndrome exchange mechanism includes

an option to select a preference for data availability or confirmed integrity.

5.2 Modeling SAFEbus Using AADL

Figure 3 shows the AADL model for the SAFEbus fault-tolerant architecture. The AADL model consists of five Line Replaceable Module (LRM) (figure shows only two). The LRMs communicate through a dual, self-checking bus pair.

The LRM is modeled as an AADL system, as shown in Figure 4. Each LRM consists of a host, two Bus Interface Units (BIUs), and four BTLs components. Both the host and the BIUs implement a self-checking pair (SCP), respectively.

The host consists of two hostDevices. The hostDevices communicate through two bridgeComparatorDevices. We use the AADL device concept for all these subcomponents, as they are implemented in the hardware. The bridgeComparatorDevices implement the SCP functionality by comparing inputs from both hostDevices. If the inputs do not match, the SCP goes silent and produces no output.

If the data from the two hostDevices matches, the bridgeComparatorDevices write the data in the interModuleMemory through an on-chip interconnect. The interconnect connects the host and BIUs through AADL bus access connections, modeling hardware connectivity.

For modeling purposes, the BIUs are broken up into two devices. The busInterfaceUnitHostDevice represents the host-side functionality of the BIU, and the busInterfaceUnitNetworkDevice represents the network-side functionality of the BIU. Both devices model dedicated hardware units. The two BIUs implement an SCP, and thus exchange data connections.

The BIUs are connected to BTLs that relay data to the two self-checking bus pairs. BTLs are simple hardware devices.

5.2.1 Bus Access vs. Data Connections for Buses

We modeled the self-checking bus pair using four busDevices that are connected to the BTLs through regular data connections rather than through AADL bus access connections. Many factors contributed to this decision:

- To bind data flows to the buses, dependencies between the host applications must be specified, and therefore the LRMs. To make the analysis generic, the only dependency we could assume is between the LRMs and the self-checking bus itself.
- We wanted to capture the SCP behavior of the BIUs and buses. Thus, we had to explicitly model the event flow from the host down to the BTLs and between the BTLs.
- The SAFEbus buses are self-checking pairs and therefore have an active role in error mitigation. AADL buses require data flows to be bound to them. We found it hard to identify data flows originating from the host, traveling to the BTLs, then propagating back up on the receiving side. Expressing the self-checking behavior of hosts and BIUs requires mapping all possible paths to the bus, but would still not properly express the way events are exchanged between components for checking purposes. In the end, we found it simpler to explicitly specify the data flow through data connections.
- Devices can source errors in the EDICT tool suite, but buses cannot. We were not able to model SAFEbus in EDICT properly due to the lack of voting logic and complex cross-communication between BIUs.

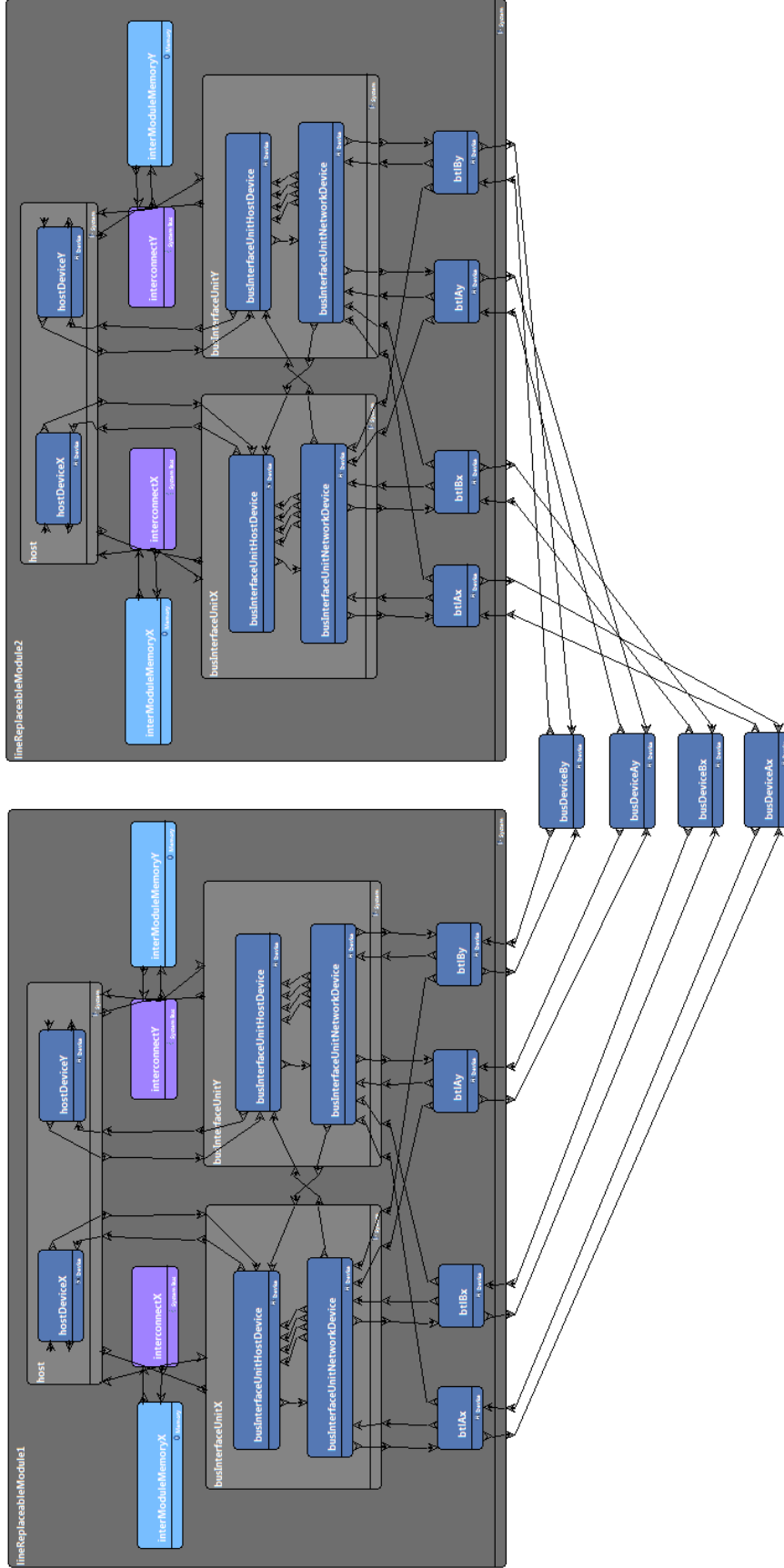


Figure 3. AADL Model of SAFEbus

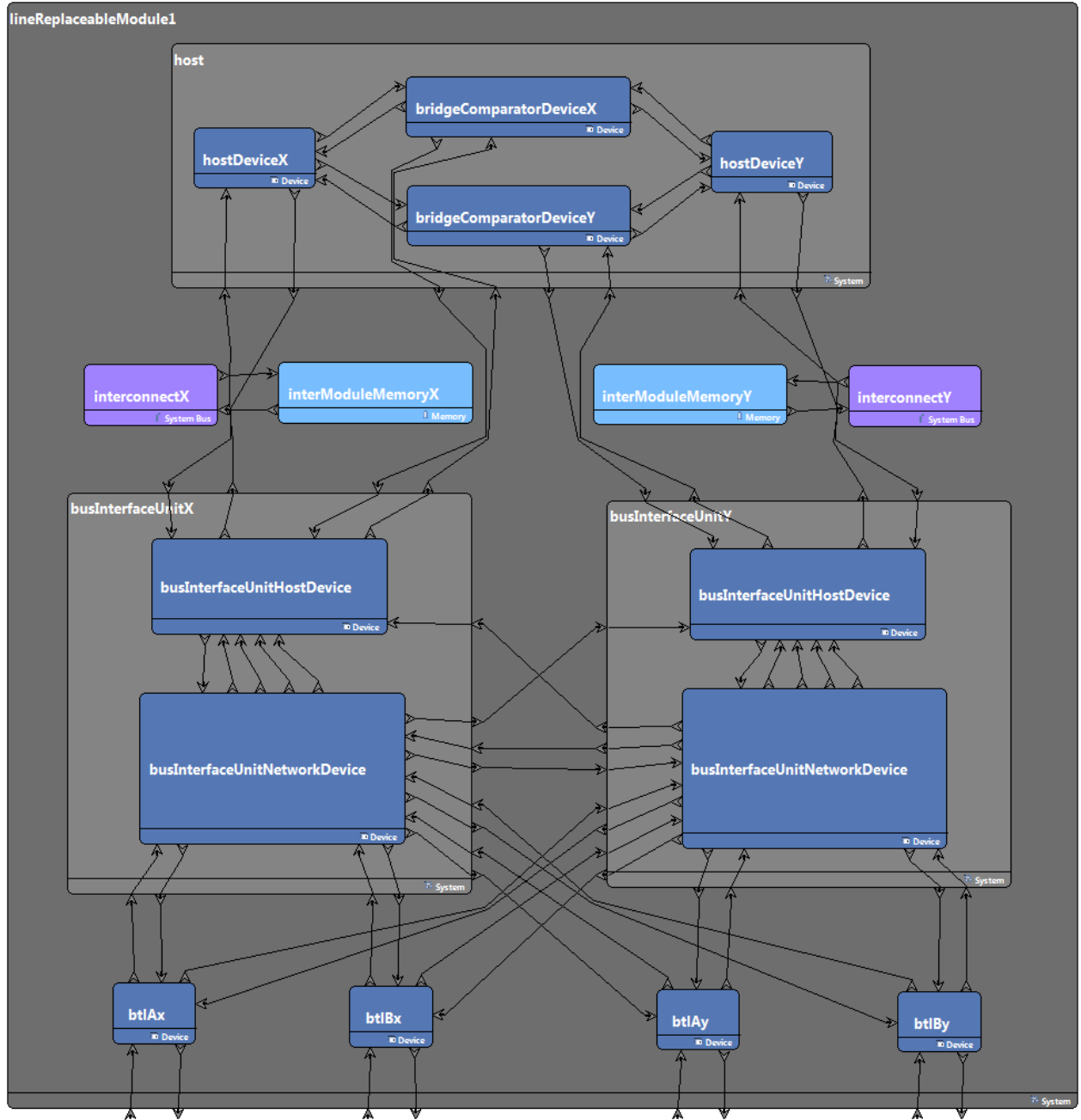


Figure 4. AADL Model of a SAFEbus Line Replaceable Module

5.3 Modeling Error Propagation in SAFEbus Using the AADL Error Annex

This section describes how we utilized the AADL Error Annex to model error propagation and mitigation in the SAFEbus architecture. These models express the error propagation in the models, but do not specify mitigation. In Sections 5.3.6 and 5.3.7, we describe how we modeled error mitigation using guards.

5.3.1 Host Error Model

The host model starts in the **s_errorfree** initial state and consists of the following states:

s_o representing fail-stop omission.

s_tsd representing timing errors. This type of error is symmetric and detectable.

s_van representing value asymmetric error condition. This error is not detectable.

s_vsn representing value symmetric error condition. This error is also not detectable.

We did not model explicitly permanent and transient error states for SAFEbus. All the error states are transient and can return to the **s_errorfree** state with a given probability. We decided on this method to simplify the overall model. Practically, all error states can have a transient and a permanent manifestation. An additional transient state is needed to properly model conditions such as when the errors occur with some probability and when specific errors are permanent while others are transient.

We intentionally kept the model simple by not modeling transient and permanent error states. We will reevaluate the practical advantages of separating states, error events, and error propagations based on persistence when applying formal analysis to the error models.

The model can exhibit four types of faults:

e_o representing a fail-stop fault event.

e_tsd representing timing fault events. This event models a symmetric and detectable error event.

e_van representing value asymmetric fault events. This fault event is not detectable.

e_vsn representing value symmetric fault events. This fault event is not detectable.

All faults arrivals follow Poisson distribution parameters. The model can propagate the following errors:

p_o representing a fail-stop error propagation.

p_tsd representing timing errors. This event models a symmetric and detectable error event.

p_van representing value asymmetric error propagation. This error is not detectable.

p_vsn representing value symmetric error propagation. This error is not detectable.

The host AADL Error Annex model implementation describes transitions between error states as a function of error events and error propagations.

The first set of rules, under “Receiving errors” describes how the automaton moves to error states from the initial **s_errorfree** state as a result of receiving incoming error propagations. The next set of rules, under “Sourcing errors,” demonstrates how the automaton sources error propagation events. Once in an error state, the component continues to propagate errors corresponding to that state. The rules under “Fault events leading to errors” describe transitions that lead the automaton from the **s_errorfree** initial state to error states corresponding to the fault events sourced by the component. Finally, under “Recovering from errors,” the rules specify how the automaton may recover from transient error states when repair events occur probabilistically.

5.3.2 BridgeComparator Error Model

The BridgeComparator implements the Self-checking Pair (SCP) behavior of the host. It explicitly encodes voting on the automaton’s inputs. The automaton starts in the **s_errorfree** state and moves to Miscompare whenever it receives an input error propagation from either of the hostDevices. When in the Miscompare state, it will propagate a **NO_MATCH** error propagation to both hostDevices and shut down all traffic toward the BIU.

5.3.3 BusInterfaceUnit Error Model

The `busInterfaceUnitHost` specifies the host-side functionality of the BIU. It starts in the `s_errorfree` initial state and has the following associated error states:

`s_o` representing fail-stop omission.

`s_tsd` representing timing errors. This type of error is symmetric and detectable.

`s_van` representing value asymmetric error condition. This error is not detectable.

`s_vsn` representing value symmetric error condition. This error is not detectable.

`s_b` representing a babbling error state.

Just like the host, it has error events and error propagations associated with each error state. Furthermore, it can receive a `NO_MATCH` error event from the `BridgeComparator`, representing an omission error.

The `busInterfaceUnitNetwork` specifies the network-side functionality of the BIU. It has the same error states, error events, and error propagations as `busInterfaceUnitHost`. The only difference is in the mitigation strategies, as described in Sections 5.3.6 and 5.3.7.

5.3.4 BackplaneTransceiverLogic Error Model

The BTL has the following error states: `s_errorfree`, `s_o`, `s_van`, `s_vsn`, `s_b`. These include all errors of the BIU except for timing errors because the BTL logic is pretty simple and we felt it could not source timing errors on its own. While the BTL could propagate timing errors through, this does not happen because the BTL mitigates timing errors on the sender side.

5.3.5 BusDevice Error Model

The `BusDevice` represents the communication bus medium of `SAFEbus`. It has the same error states, error events, and error propagations as the BTL. Since the bus is just a wire, it cannot source timing errors, but it can contribute to value errors or babbling due to link failure.

5.3.6 Error Mitigation Modeling - Sender Side

The error propagation and mitigation logic of `SAFEbus` is modeled exclusively using AADL guards, introduced in Section 2.2.2.

We did not have to rely on guards to specify the SCP behavior for the host. The `bridgeComparatorDevices` will send `NO_MATCH` error propagations to the BIUs whenever the data comparison fails. The `NO_MATCH` error represents an omission error.

Whenever the BIU receives omission errors or `NO_MATCH` error propagations, it propagates through an omission error. There is no associated error state in any of the error automata. Value asymmetric and value symmetric errors directly propagate through the BIUs on the sender side. These are mitigated at the receiver side.

The `bridgeComparatorDevices` write the `interModuleMemories` directly, and the BIUs receive data from the host by reading the written data from the shared memory. This implementation essentially transforms timing errors into value errors; if the host and BIU do not write and read the shared memory in sync, the BIU may read bad data from the memory. We expressed this error transformation by putting guards on the BIU side.

The `busInterfaceUnitNetwork` propagates through all error propagations arriving from the `busInterfaceUnitHost`, except for timing errors, which are mitigated at the sender side. The BTLs require both the data from its respective BIU and an enabling event from the other BIU.

When a BIU (`busInterfaceUnitX`) sends data to its BTL, it also sends an enabling event to the other BIU (`busInterfaceUnitY`). There is an AND gate inside `busInterfaceUnitY` where the other BIU can set a validation indicator. Then the `busInterfaceUnitY` bounces the enabling event back to the BTLs of `busInterfaceUnitX`. Thus, if timing is off, the BTLs will not receive the enabling event and the data is not sent, but instead the LRM goes silent.

We did not need to introduce additional modeling to capture this activity. We simply did not add propagation for timing errors in the BTLs, thus timing errors never get out of the BIUs.

5.3.7 Error Mitigation Modeling - Receiver Side

On the receiver side, no mitigation occurs in either the `busDevices` or the BTLs. The BIUs are the first line of mitigators in the LRMs. Each `busInterfaceUnitNetworkDevice` will perform a comparison on the following inputs arriving from BTLs: $Ax: Ay$, $Ax: By$, $Bx: Ay$, $Bx: By$. It will then send the results of the four comparisons upstream to the `busInterfaceUnitHostDevice`. It also encodes the result of these comparisons and whether it has received input from each bus; it transmits this data to the other BIU by sending it to the `busInterfaceUnitHostDevice` of the other BIU.

In the next step, each `busInterfaceUnitHostDevice` has the result of the vote of its own `busInterfaceUnitNetworkDevice`, as well as the result of the vote from the other BIU's `busInterfaceUnitNetworkDevice`. It then performs a table lookup based on the data it received to figure out how to mitigate the various types of potential error combinations.

We did not model all of the table lookup in detail. (In the ARINC 659 standard, these tables run on for 11 pages.) We simplified the model by specifying that all single errors are tolerated and no multiple errors are tolerated. Multiple errors are translated into an omission error, as the BIUs fail silently.

Finally, the `bridgeComparatorDevices` receive inputs from the two BIUs and perform a comparison. Thus, if any error occurs on any of the inputs, for error modeling purposes we assumed that the comparison fails. The `bridgeComparatorDevices` send a `NO_MATCH` error propagation to the hosts, so in this case they are made aware that the comparison failed at the host-side SCP.

6 Case Study: BRAIN

6.1 Protocol Overview

The Braided Ring Availability Integrity Network (BRAIN) is a novel communication architecture supporting fault-tolerant, time-triggered communication. As the name suggests, the BRAIN is built on a braided-ring topology. This topology augments the standard ring topology with increased connectivity. In addition to the “direct link” connections between a node and its immediate neighboring nodes (as is used in simple rings), a braided-ring node is also connected to its neighbor’s neighbor via a link called the braid or skip link (see Figure 5). The BRAIN utilizes the additional connectivity to achieve both high-coverage integrity and availability concurrently. The BRAIN can use almost any existing local area network (LAN) technology to implement its communication links, including any of the IEEE 802.3 Ethernet variants. The BRAIN uses the least amount of hardware to achieve single fault tolerance (including Byzantine failure) of any known data network. The BRAIN can tolerate most cases of two benign faults with no additional redundancy. The BRAIN topology enables adjacent nodes to collaboratively form SCPs. This allows standard simplex computational hardware to be run-time configured into high-integrity fail-silent computational platforms, which provides the high fault coverage for processing that one would find in architectures supported by SAFEbus but without requiring any special SCP hardware for the processors. The BRAIN’s benefits derive from its time-triggered data flow and its use of high-coverage fault tolerance.

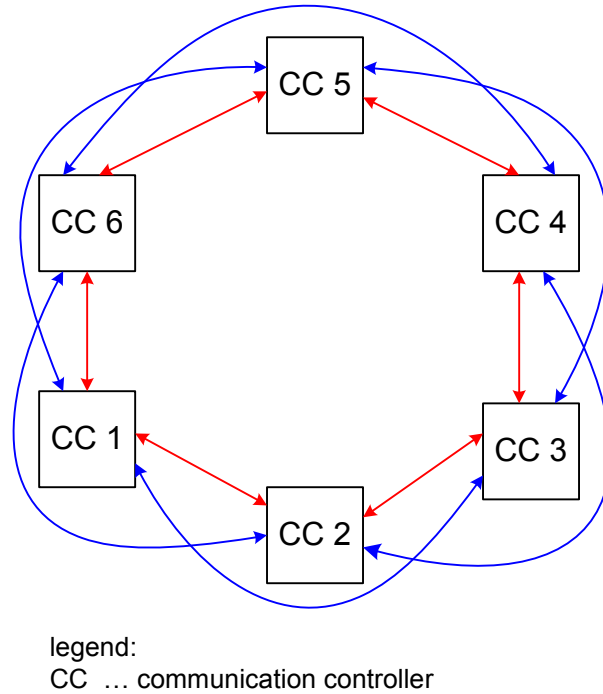


Figure 5. BRAIN’s Braided Ring Basic Architecture

The BRAIN uses the following protocol mechanisms and capabilities:

- **Self-Checking Data Relay** mode focuses on inline integrity failure detection that detects any possible corruption of data as it is being relayed. As data is transported around the ring, each node compares the data it receives on the skip and direct links. If the data miscompares, the loss of data integrity is marked using a field appended to the message. Normally, the data

from the skip link is selected for forwarding. In the BRAIN, the sending node transmits its message in both directions around the ring. Broadcasting a message in both directions provides availability, since a message will be delivered successfully if either one of the directions is intact. Given a single-fault assumption, the independence of these two paths ensures that one successful path will always be available from any arbitrary sending node to any arbitrary receiving node.

- **Independent Path Data Integrity Reconstitution** focuses on tolerating a second benign (fail-stop or omission) failure. The BRAIN can tolerate a benign second fault without any increase in redundancy, which provides an additional degree of fault tolerance. To implement this tolerance, each receiving node compares all of the data it received from one direction with the data it received from the other direction. If data received each direction is bit-for-bit identical, the data integrity may be reconstituted and the data used even if either or both of the inline data integrity markers indicate loss of integrity. However, the current evolution of the BRAIN cannot tolerate an active fault and an arbitrary benign fault at the same time. The first propagation mode provides for fail-op/fail-stop operation. The second propagation mode adds fail-op/fail-op/fail-stop operation for benign faults; this is worst-case. The BRAIN can tolerate many more faults for most cases. For example, the BRAIN can tolerate any number of benign node failures, as long as two or more failed nodes are not adjacent when three or more nodes have failed.
- **Self-Checking Processor Pair Broadcast:** The BRAIN's connectivity and data relaying policies can be used to compare the output of two adjacent nodes. This comparison allows for adjacent nodes to be configured into high-integrity message-based self-checking pairs. Implementing the paired actions is as simple as configuring the communication schedule to make the two halves of a pair transmit in a shared slot (the time allocated on the media to transmit one message). The synchronous nature of the BRAIN and the high-integrity forwarding mechanism ensure that the receiving nodes receive a single high-integrity message when the data sent from the two halves of the pair are identical.
- **Time-Triggered Sequenced Guardian Roles** provide additional mechanisms to qualify data as it enters the BRAIN and ensure that the BRAIN's data integrity is consistent for all member nodes. These guardian roles cross check and police data as it enters the BRAIN. The specific roles are selected in accordance with the Time Division Multiple Access (TDMA) schedule and are performed by the active transmitting nodes' immediate neighbors (direct links) and neighbors' neighbors (skip links). Hence, it is called Brother's Keeper Guardianship. Note that the guardian, being an independent neighboring node, ensures that guardian action is fully independent of the transmitter it is guarding, which gives all the benefits of fully independent redundant guardian hardware without requiring the addition of any redundant hardware components.

For further detail, refer to Chapter 5 of [8].

6.2 Modeling BRAIN Using AADL

Figure 6 shows the AADL model for the BRAIN. The figure shows five BRAIN nodes interconnected with each other using dual-channel ports. BRAIN nodes are shown in Figure 7.

Each brainNode is modeled as an AADL component and is composed of a brainNodeBusInterfaceUnit BIU subcomponent, a brainNodeQuadBusDriver subcomponent, a brainNodeHost sub-

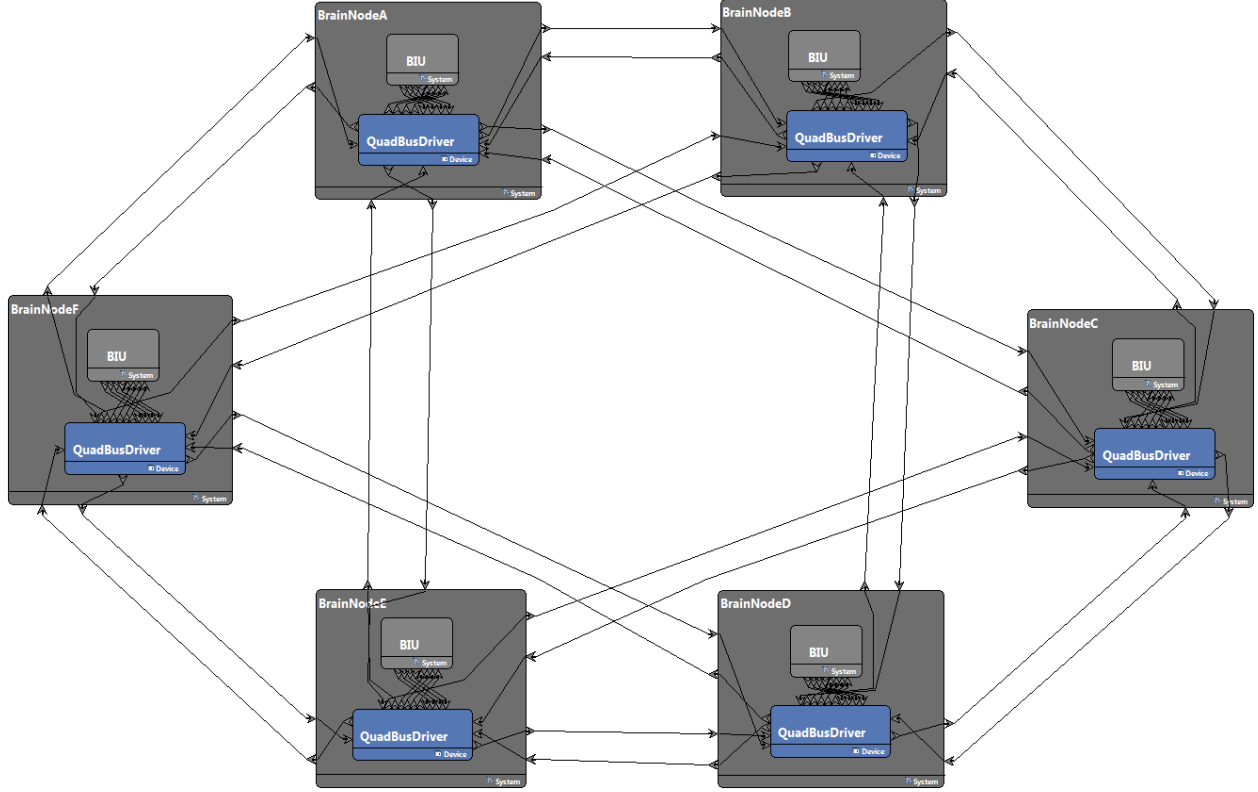


Figure 6. AADL Model of BRAIN

component, a local memory, and a system bus (interconnect) that interconnects the memory with the BIU and the brainNodeHost.

The brainNodeQuadBusDriver subcomponent is made up of four AADL devices, each of which models a bus driver and connects its node with another brainNode. Both devices model dedicated hardware units.

The brainNodeHost subcomponent consists of one AADL processor model with local memory (brainNodeProcessor) and one AADL process model (brainNodeProcess). brainNodeProcess has one AADL thread (brainNodeThread) that has a direct data connection with the BIU.

6.2.1 Bus Access Connections

We did not model bus access connections for the BRAIN. Each BRAIN node connects to other BRAIN nodes using point-to-point connections; thus, there is no shared bus concept. Moreover, at this stage, it is not fully evident how the hardware platform will be utilized for BRAIN. Finally, regular data connections captured all the dependencies required for the formal analysis of the error propagation and mitigation.

6.3 Modeling Error Propagation in BRAIN Using the AADL Error Annex

We selected the BRAIN as a modeling candidate because of its unique mechanisms for integrity qualification. As outlined in Section 6.1, the nodes of the BRAIN compare data they receive on skip and direct links as part of data propagation. The BRAIN flags comparison error data as question-

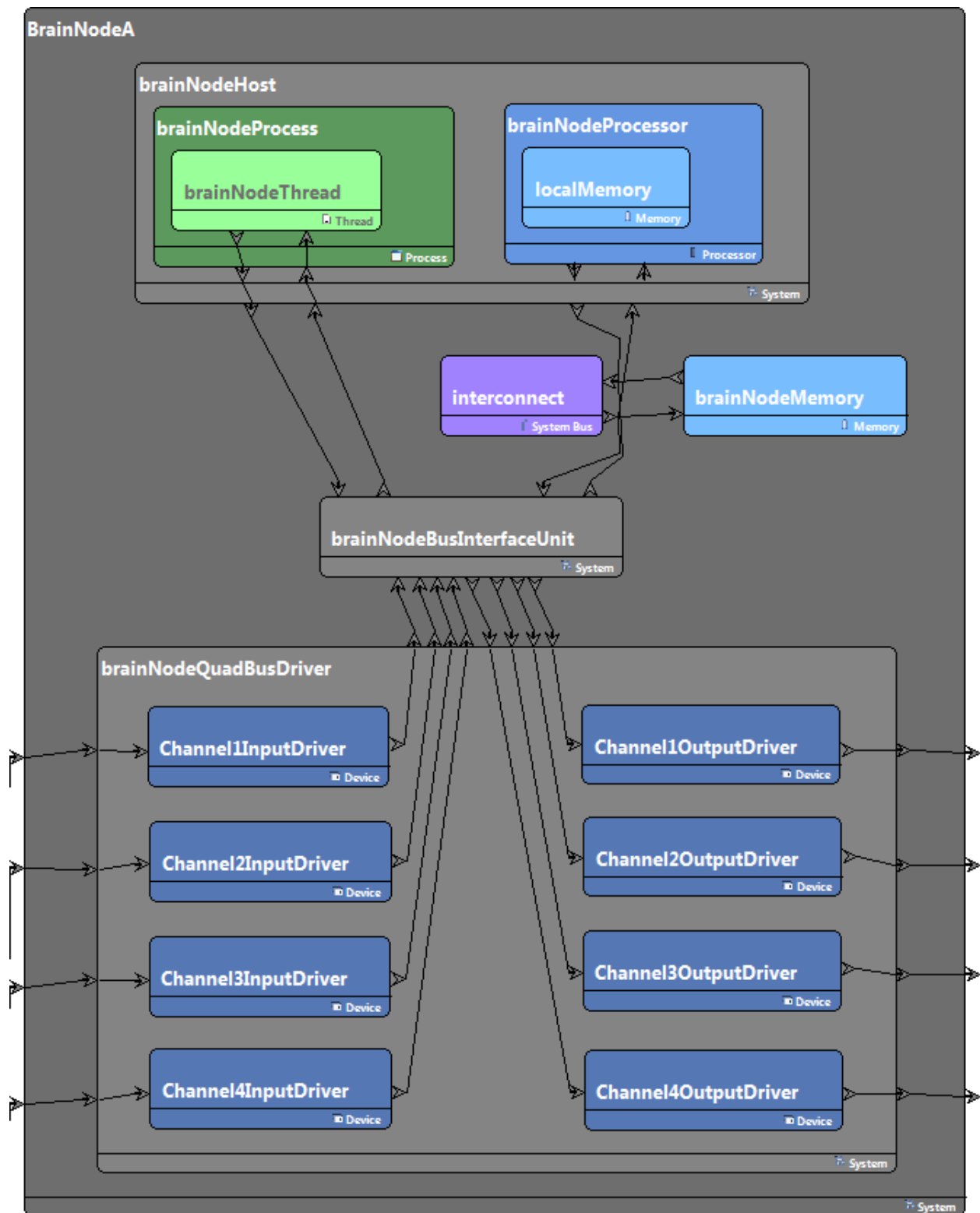


Figure 7. AADL Model of a BRAIN node

able or untrusted. The comparison is implemented such that, once a data stream is identified as questionable, it cannot be marked good by any downstream propagating node.

The BRAIN fault tolerance is built on the assumption that only one node will actively fail. It also assumes that all second faults are restricted to passive connectivity reduction. Ideally, the connectivity path reduction will be captured within all components that can contribute to such failures; however, in the initial model, we have limited the `brainNodeBusInterfaceUnit` to symmetric error manifestations across its interfaces.

The system-level influence of connectivity asymmetry is captured in the behavior of the `brainNodeQuadBusDriverSystem` component, which is split into eight subcomponents and has separate error state machines at each interface, similar to the TTP driver.

The driver model for the BRAIN is simplified. In the initial model, the driver is assumed to emit only omission, babble, or detectable errors, and we assume that the driver is unable to create semantic content. In addition, since the BRAIN nodes are connected through point-to-point connections, asymmetric value errors are not assumed as disagreement is not possible. Non-congruent `p_m_nc` error propagation is added to the `brainNodeBusInterface` component at the higher level to capture an erroneous mode of the `brainNodeBusInterface` to send different values on its outgoing links. Thus, the system-level impact of value arbitrary errors is still captured within the model.

Currently, all failure modes are assumed to have an identical probability, but separate event transitions are included to allow a more granular investigation of the behavior as the analysis tools become available. To simplify the data propagation modeling, we added the concept of an error-free propagation `p_err_free` to the model. This concept is implemented by nodes querying the ingress links for the `err_free` start, and following propagation is modeled with an explicit propagation `p_err_free`. A questionable error-free propagation `p_err_free_q` was also included to capture `err_free` flows that did not match on the accompanying skip or direct link during propagation. With these constructs in place, the basic data integrity modeling of the BRAIN became relatively simple. The included model is much simpler than earlier instantiations of the same model without these concepts. The guards have these basic conditions:

- To propagate error-free data (`p_err_free`) when good data is present on skip and direct links.
- To propagate questionable error-free `p_err_free_q` when good data is present on one link and the other link is empty `p_o`.
- To propagate questionable error-free data `p_err_free_q` when questionable `p_err_free_q` data arrives on one link and unquestionable error free data `p_err_free` is present on the partner link.
- To propagate omission `p_o` when both the links of the ingress direction are empty.
- To propagate a value fault flagged as questionable `p_vq` for other ingress fault combinations.

A directional propagation `out` guard is mapped for each direction to capture the basic error propagation properties of the BRAIN data relay at a summary level. The model also includes a guard for the host consumer. This guard implements the receive data acceptance tests of the BRAIN. This test masks errors if a minimal error-free propagation path exists. The tests correspond to:

- Receiving error-free data `p_err_free` on both skip and direct links from either direction.
- Receiving at one sample of either error-free data `p_err_free` or questionable error-free data `p_err_free_q` from each direction.

6.3.1 Limitations of the Initial BRAIN Model

Although we believe that the description above captures the basic error propagations of the BRAIN, we decided not to refine the model further until clarification of the detailed semantics has been

examined. Noting possible issues with the current representation and guard out semantics, we are arranging a peer review of the model with noted AADL error modeling experts.

Time-sequenced guardian and self-checking pair behavior have not yet been explicitly modeled. These mechanisms are interesting because the behavior traverses around the ring as the TDMA schedule progresses. For an example, see the directional integrity enforcement of the `p_m_nc` data propagation.

Currently, the only way to capture such behavior in AADL is to utilize mode mechanisms. Mode mechanisms seem more appropriate for higher-level software and system mode interaction, and using them may be cumbersome for detailed protocol modeling. In addition, we are unsure of the temporal impact of the directional guardian exchanges and how such exchanges are resolved within the error model execution assumptions. Hence, we are investigating different potential abstractions to aid the generic treatment of such behavior.

A key challenge we have identified for using BRAIN to model error propagation and mitigation is the need to compose multiple, potentially heterogeneous models of computation (MoC) to express the behavior of both the analyzed system and the error propagations. The current AADL Error Annex relies on a probabilistic automata context, whereas AADL itself is defined using dataflow-like semantics. For the formal analysis of error propagation in BRAIN, the composition of such models must be captured. The behavior of the BRAIN nodes themselves must also be captured, potentially through the AADL Behavioral Annex, finite state machines (FSM), or other formal languages.

The BRAIN error states have not yet been mapped to software. We intend to explore these interactions within the TTP context and apply lessons learned later, but from a conceptual viewpoint, event ports should be added to allow protocol signaling to the host processing system.

Errors from the host are also not explicitly mitigated because of issues integrating such faults with the scheduled transmission and guardian activity of the protocol. Once a suitable representation has been devised, the models will be updated accordingly.

7 Case Study: SPIDER

7.1 SPIDER/ROBUS Protocol Description

The Reliable Optical Bus (ROBUS) is the key component and the core communication network of the Scalable Processor-Independent Design for Enhanced Reliability (SPIDER) system (see Figure 8, a general-purpose fault-tolerant integrated modular architecture developed at NASA Langley Research Center. The ROBUS is a TDMA broadcast communication system with a time-indexed communication schedule. The ROBUS services include message broadcast (Byzantine agreement), dynamic communication schedule update, clock synchronization, and distributed diagnosis (group membership). The ROBUS also features fault-tolerant startup and restart capabilities. See [9] for more information.

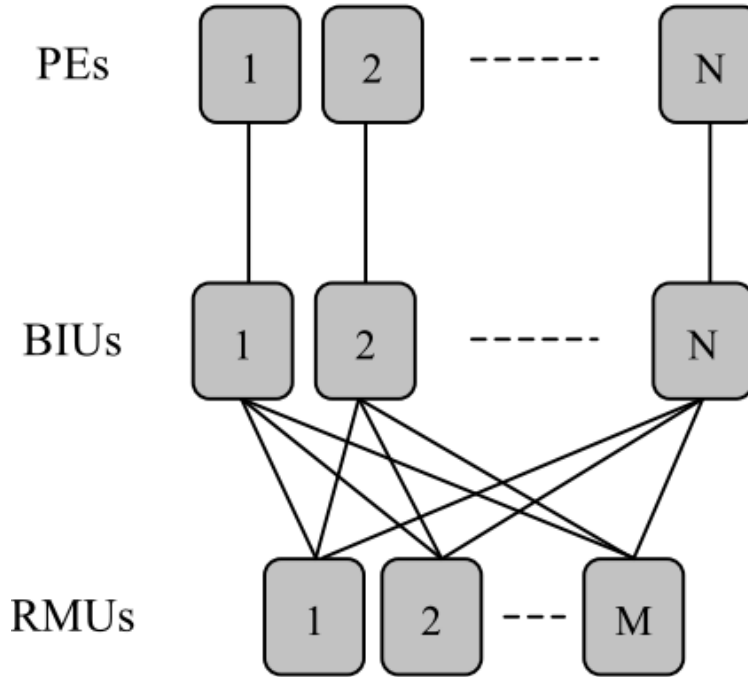


Figure 8. The ROBUS Network of SPIDER

7.2 Modeling SPIDER Using AADL

Figure 9 illustrates the high-level AADL model of the SPIDER fault-tolerant architecture. SPIDER consists of a set of Processing Elements (PEs) communicating through a ROBUS. The ROBUS consists of redundant BIUs and Redundancy Management Units (RMUs). Figure 9 illustrates a ROBUS consisting of three BIUs and three RMUs (referred to as 3×3 ROBUS). The ROBUS, PEs, BIUs, and RMUs are all modeled as AADL systems.

The PE is composed of the following components. The `processingElementBusDriver` is an AADL device that models the hardware unit driving read/write communication towards the ROBUS. The `processingElementProcessor` models the Central Processing Unit (CPU) that serves as the execution platform for host-side applications. The `processingElementProcessor` reads and writes data to the `processingElementMemory` through an on-chip interconnect. The `processingElementBusDriver` uses the same shared memory to relay and receive messages to and from the ROBUS.

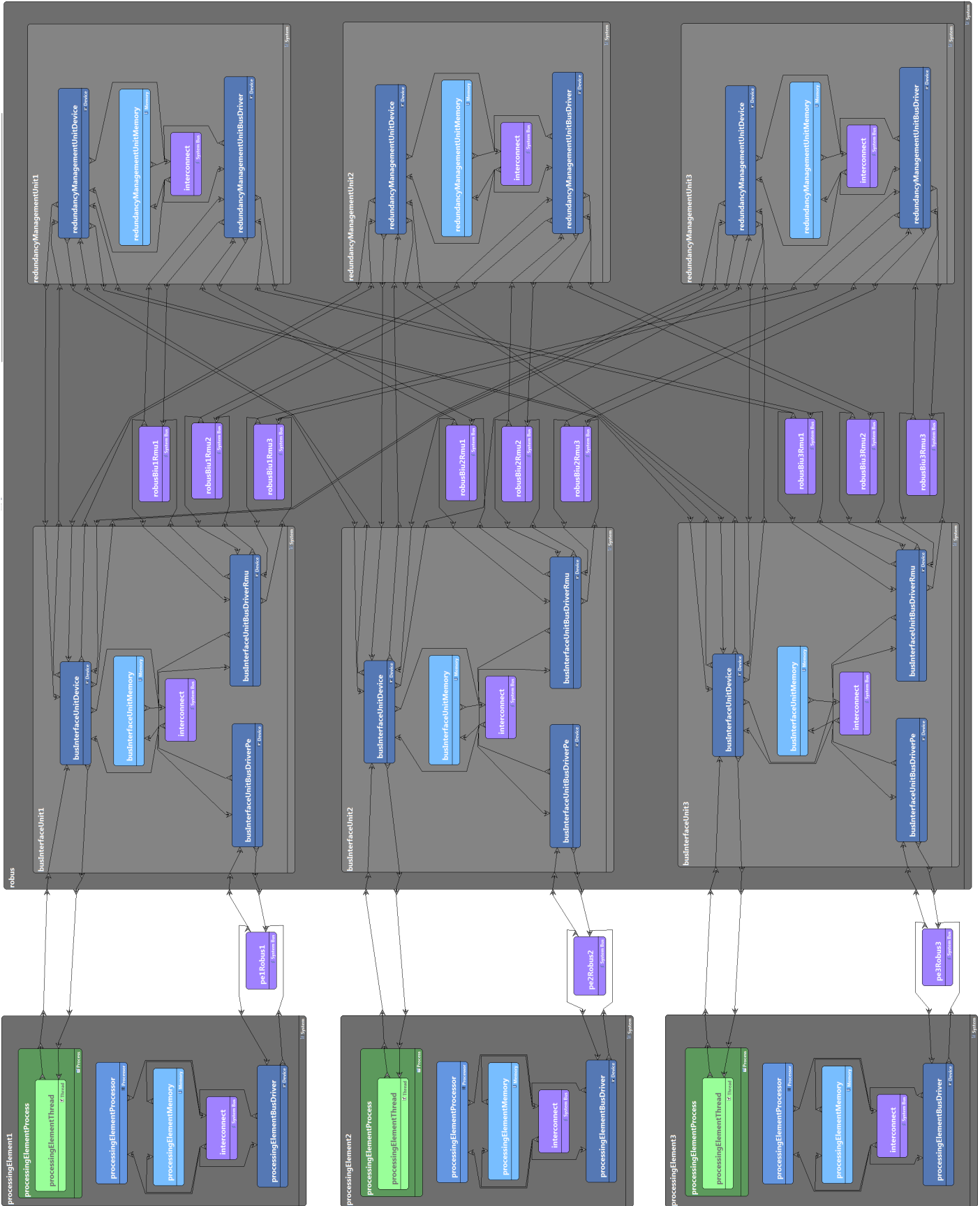


Figure 9. AADL Model of SPIDER

The `processingElement` also includes a `processingElementProcess`. The `processingElementProcess` contains a `processingElementThread`. This thread models the host-side software application. The thread and process are bound to the `processingElementProcessor` using AADL constructs. The `processingElementThread` is connected to the ROBUS through data connections. The hardware connection between the PEs and the ROBUS is modeled as a bus access connection between the `processingElementBusDriver` and the ROBUS, propagating through the `peXRobusY` bus components.

The PEs access the ROBUS through BIUs, which are modeled using the following components. The `busInterfaceUnitBusDriverPE` and `busInterfaceUnitBusDriverRmu` devices model the hardware components managing data communication through buses toward the PEs and RMUs, respectively, and write data in the `busInterfaceUnitMemory` memory component through an on-chip interconnect. The `busInterfaceUnitDevice` component models the hardware unit that implements the SPIDER protocol services, including voting, etc.

In the ROBUS, BIUs and RMUs form a fully-connected bipartite graph. All BIUs are connected to all RMUs, and vice versa. Figure 9 models a 3×3 ROBUS, so all BIUs have three outgoing bus access connections that connect the `busInterfaceUnitBusDriverRmu` components to the RMUs through `robustBiuXRmuY` bus components. Likewise, data connections between the `busInterfaceUnitDevices` and RMUs are captured using AADL data connections.

In RMUs, the `redundancyManagementUnitBusDriver` component models the hardware unit responsible for managing communication with the BIUs, modeled as AADL bus connections. The `redundancyManagementUnitDevice` models the hardware component implementing Rmu-side protocol services, such as reflecting BIU messages back to all other BIUs. Similar to PEs and BIUs, communication between the `redundancyManagementUnitDevice` and `redundancyManagementUnitBusDriver` components is managed through a shared `redundancyManagementUnitMemory`, accessed through an on-chip interconnect by both devices.

7.3 Modeling Error Propagation in SPIDER Using the AADL Error Annex

We used the AADL Error Annex to analyze error propagation and mitigation in the SPIDER architecture. We did not model either the SPIDER synchronization algorithm or the operational modes related to clique detection/initialization. The current AADL models are an abstraction of the SPIDER protocol, focusing on error propagation on a synchronous platform and related mitigation strategies. We defined error models for the three major component types: the PEs, BIUs, and RMUs. The following subsections describe these error models in detail.

7.3.1 Processing Element Error Model

The PE model starts in the `s_errorfree` initial state and consists of the following states:

`s_o` representing fail-stop omission.

`s_pe_local` representing PE local errors.

`s_vsn` representing value symmetric error condition. This error is not detectable.

`s_link_corrupt` representing an error case corresponding to a failed link between the PE and the ROBUS.

`DETECTED_ERROR` representing a state when the PE detects an error it has received.

We did not model explicitly permanent and transient error states for PE states. All the error states are transient and can return to the `s_errorfree` state with a given probability. We decided on this to simplify the overall model. Practically, all error states can have a transient and permanent manifestation. Then one needs to add an additional transient state, to properly model conditions

such as when the errors occur with some probability, and a certain percentage of said errors is permanent while the other percentage is transient.

To keep the model simple, we did not model transient and permanent error states. We will re-evaluate the practical advantages of separating states, error events, and error propagations based on persistence when applying formal analysis to the error models.

The model can exhibit four different faults:

e_o representing a fail-stop fault event.

e_pe_local representing PE local fault events.

e_vsn representing value symmetric fault events.

e_link_corrupt represents a fault event corresponding to a failed link between the PE and the ROBUS.

We did not include value asymmetric errors for the PEs, as they are connected to only one BIU, so every value error is essentially symmetric. All fault arrivals follow Poisson distribution parameters. The model can propagate the following errors:

p_o omission errors. These represent events where a data is absent. This can either be detected or not.

p_pe_local PE Local error propagation.

p_vsn Value symmetric error propagation. This type of error is not detectable.

p_link_corrupt Link Corrupt error propagation represents a broken link between the PE and the ROBUS.

The PE AADL Error Annex model implementation describes transitions between error states as a function of error events and error propagations.

The first set of rules, under “Receiving errors,” describes how the automaton moves to error states from the initial **s_errorfree** state as a result of receiving incoming error propagations. The rules under “Sourcing errors” demonstrate how the automaton sources error propagation events. Once in an error state, the component continues to propagate errors corresponding to that state. Rules under “Fault events leading to errors” describe transitions that lead the automaton from the **s_errorfree** initial state to error states corresponding to the fault events sourced by the component. Finally, under “Recovering from errors,” the rules specify how the automaton may recover from transient error states when repair events occur probabilistically.

7.3.2 BusInterfaceUnit Error Model

The BIUs serve as the interface for the ROBUS and also perform mitigation and error detection.

s_o representing omission error conditions.

s_tsd representing symmetric timing errors.

s_van representing value asymmetric undetectable errors.

s_vsn representing value symmetric undetectable errors.

s_header_corrupt representing error states corresponding to corrupt headers in messages received.

s_link_corrupt representing error states corresponding to corrupt communication links.

The BIU associates an error event and error propagation with each error state. Its error propagations are defined as follows:

NO_MAJORITY a consensus could not be reached during voting.

PE_ERROR the BIU detects an erroneous PE.

SOURCE_ERROR either of the BIUs or RMUs sending messages to the BIU are detectably erroneous.

7.3.3 RedundancyManagementUnit Error Model

The RMU error model is similar to the BIU error model. It consists of the `s_errorfree`, `s_o`, `s_tsd`, `s_van`, `s_vsn`, `s_header_corrupt`, and `s_link_corrupt` states. It also includes the `PE_ERROR` and `SOURCE_ERROR` error propagations. It does not, however, propagate `NO_MAJORITY` errors, as RMUs are not performing Triple Modular Redundancy (TMR) voting on multiple-PE inputs.

7.3.4 Error Mitigation Modeling - Sender Side

We modeled error detection on BIUs by introducing mappings between error propagation events. The BIU detects omission (`p_o`), `PE_Local` (`p_pe_local`), and `link_corrupt` error propagation and transforms them into a `PE_ERROR` propagation.

On the RMUs, all `PE_ERRORS` are propagated back to all BIUs. Any other detected errors will lead to a `SOURCE_ERROR` error propagation back to the BIUs.

7.3.5 Error Mitigation Modeling - Receiver Side

Although all the PEs, BIUs, and RMUs perform error detection, actual error mitigation is performed at the BIUs after they receive inputs from all the RMUs.

If more than two incoming value error propagations occur in the BIU, the voting cannot reach a consensus, and a `NO_MAJORITY` error propagation is generated. Two or more incoming error propagations of type `p_o`, `p_tsd`, `p_header_corrupt`, `p_link_corrupt`, `PE_ERROR`, or `SOURCE_ERROR` will result in a `SOURCE_ERROR` output toward the PE, as the BIU cannot reach a consistent state. Single errors of any type are mitigated successfully by the TMR voting on the BIUs.

8 Case Study: TTP

8.1 TTP Protocol Description

TTP [10] was designed for safety-critical transportation systems (automotive, aerospace, railway) [11] and was originally intended to be a low-cost communication platform for full-authority, hard real-time, x-by-wire control applications [12]. Developed in the mid-1990s, TTP is a fully deterministic protocol implementing a strictly time-triggered communication model. In TTP, each node is allocated access to the network in accordance with a static a priori configured TDMA schedule. Each slot sends in a predetermined order once per round. The system communication cycle comprises a number of these fixed communication rounds. In TTP, the size of the TDMA slots allocated to each node may be different; however, a node's slot size per round must be consistent throughout the cluster cycle.

TTP incorporates several mechanisms to maximize network bandwidth efficiency. One key design decision affected by this drive for network bandwidth efficiency is the TTP implementation of its group membership protocol. Designed to enforce that all nodes maintain an agreed-upon view of the global communication state, TTP group membership requires all nodes to be in agreement to take part in communication. Each node maintains a membership vector that records the received status of each slot. When a node transmits, it does not send the entire vector to conserve bandwidth. Instead, the value of the membership vector is encoded into the transmitted frame's Cyclic Redundancy Check (CRC). The net effect is that nodes that have heard an agreeing set of transmissions can decode the frame correctly; however, nodes that do not agree on the membership vector cannot receive the frame content. Thus, nodes disagreeing with the global membership state are isolated into a minority clique. CRC incorporates a clique detection service that forces such nodes to reintegrate. Nodes gauge their own transmission success by monitoring their own acknowledgment as reported by the two nodes that follow them within the TTP round. Confirmation from either node is sufficient for a transmitting node to include its own transmission within the agreed membership vector content.

To allow node reintegration, the protocol also requires some nodes to explicitly transmit the membership vector (using TDMA *iframes*) at regular intervals. Later variants of the protocol also incorporate *x-frames* that allow every node transmission to contain both data and explicit protocol state.

For clock synchronization, TTP implements a fault-tolerant average (FTA) convergence function. This FTA can algorithmically tolerate a Byzantine manifestation; however, because the algorithm depends on membership implementation, the system's resilience to Byzantine failure manifestations may be compromised.

The original version of TTP implemented on-chip bus guardian functionality to contain mode failures. The guardians were conceptually simple, slot-enforcement engines; however, they suffered from logical and physical dependencies on the controller implementation. Such guardian functionality cannot be leveraged into dependability claims for real-world, certifiable systems. Fault injection experiments performed as part of the Fault Injection for TTP (FIT) project TTP also showed that the guardian was ineffective at containing Byzantine and SoS, fault manifestations [13].

The Honeywell TTP-Hub: In 2000, the protocol was selected as the backbone networking infrastructure for the Honeywell Modular Aerospace Control. The Modular Aerospace Control architecture [14] enabled reuse within its system boundary, allowing engine customization through selection of generic modules. Initially targeted at three engines, Full Authority Digital Engine Control (FADEC) architecture reuse has been very promising in this regard. The architecture

modularity and systematic redundancy management has been demonstrated to significantly reduce development schedules and nonrecurring engineering expense. As illustrated in Figure 10, the architecture incorporates additional guardian components within the FADEC boundary.

In each lane, the power-supply card hosts two bus guardians, one for each channel. The guardian functionality has been developed to address the dependability implications of the non-guarded TTP protocol described above and to implement a fully independent bus guardian function. In summary, the guardians (hubs):

- Prevent a Byzantine error from disrupting system membership.
- Prevent node masquerade failures.
- Prevent babbling failures in one lane from disrupting the other lane or system operation.
- Prevent a chronic babbling failure of one entire lane (i.e., dual-channel babbling) from disrupting the operation of the other lane.

The design of the central hub guardian is intended to be simple and suitable for implementation in a low-end programmable logic device. The rationale for this decision is that it reduces the likelihood of complex, hub-induced failure modes (where the hub creates message content). Within each lane, the connections between the nodes and each hub are point-to-point. On each channel, the hubs of each lane are connected by a transformer-coupled bus that provides galvanic isolation. To mitigate TTP value errors, the hub actively reshapes and re-times all data signals as they pass through the hub. To mitigate SoS temporal errors, the hub also actively enforces a strict temporal policy ensuring that the start-of-frames of all relayed transmissions are within a guarded tolerance, sufficient to ensure that Byzantine manifestations do not occur. To enforce these conditions, the hub must synchronize to the running cluster timeline. With limited design resources (less than 256 flip flops), implementing the SoS fault-tolerant clock synchronization is not possible; therefore, the hub incorporates a dissimilar clock parasitic synchronization approach. The hub votes *out of band signals* from the protocol controller using *action time* assertions that mark the beginning of each slot. Selection of the second *action time* signal that arrives within the expected precision tolerance is guaranteed (under a single fault assumption) to be fault-free, and the hub uses this signal as the source of its temporal enforcement timeline.

8.2 Modeling TTP Using AADL

Figure 11 shows the AADL model for a TTP architecture based on a shared dual bus. This model consists of five TTP nodes (`ttpNode`) that communicate with each other using dual-lane channels. Each `ttpNode` is modeled as an AADL system as shown in Figure 12, and the two `ttpChannels` are modeled as devices.

The `ttpNode` communicates with the dual channels through `ttpNodeBusDrivers`, modeled as AADL devices. Data from the `ttpNodeBusDriver` passes through an on-chip interconnect (`bus-DriverXBus`) to the `ttpNodeControllerSystem`. The `ttpNodeControllerSystem` subsystem comprises the `localMemory` memory module and a `ttpNodeControllerDevice` that model the TTP hardware controller. This component performs the voting between the inputs on both channels.

Each `ttpNode` includes one AADL processor (`ttpProcessor`) with local on-chip memory (`localMemory`) representing the platform for the host application. The host-side SW is modeled as an AADL process (`ttpNodeProcess`) that consists of a single thread (`ttpNodeThread`). The process and thread are bound to the `ttpNodeProcessor`.

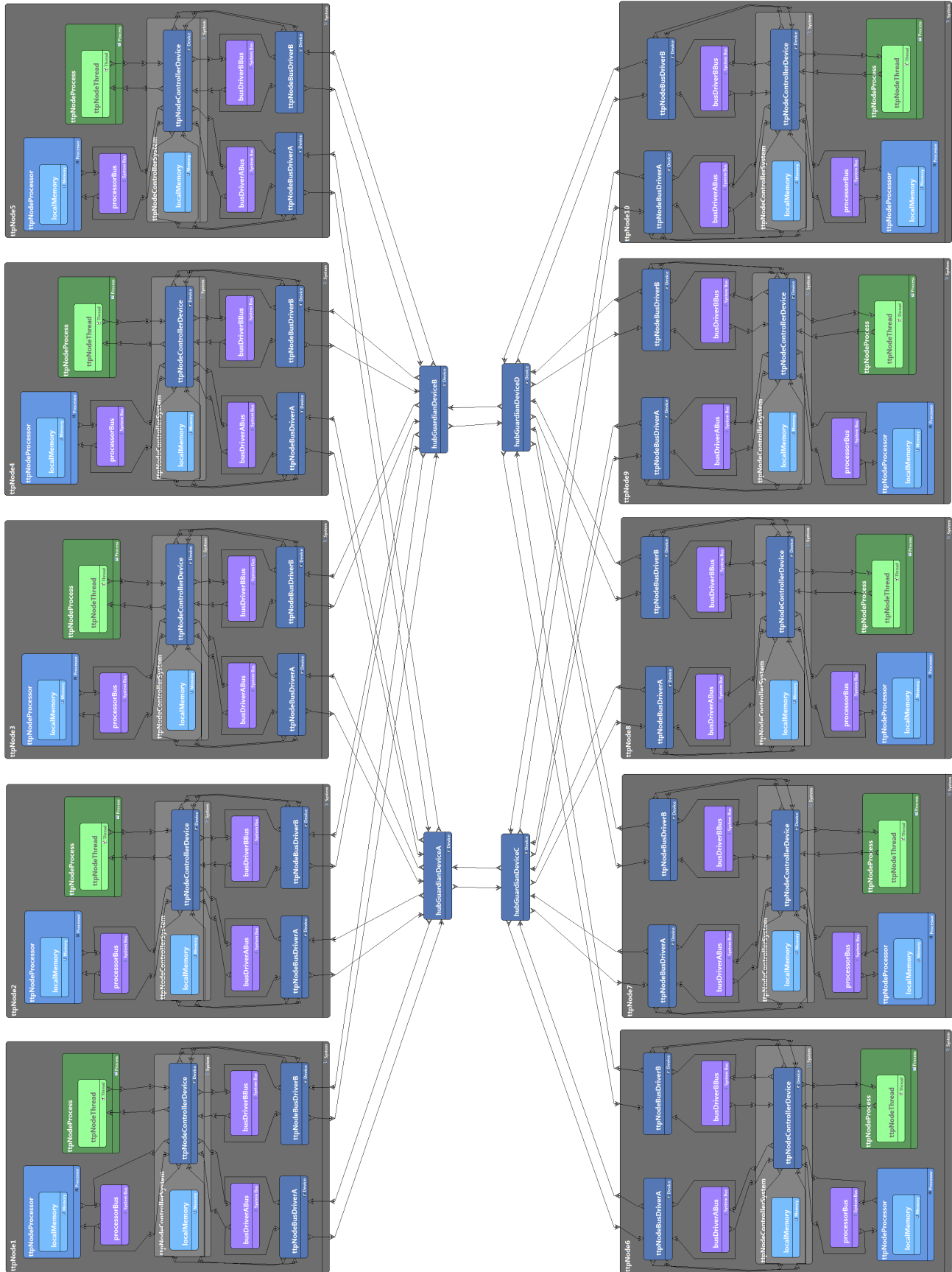


Figure 10. MAC Dual Lane Architecture based on TTP

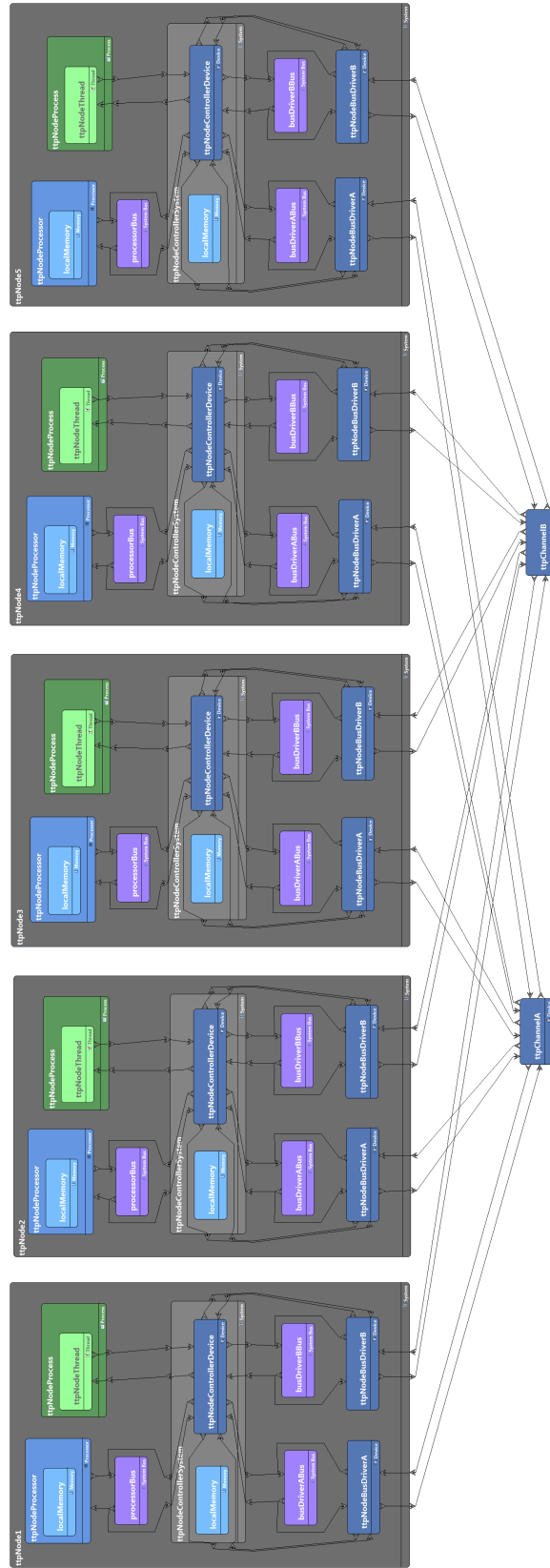


Figure 11. Dual Bus Model of TTP

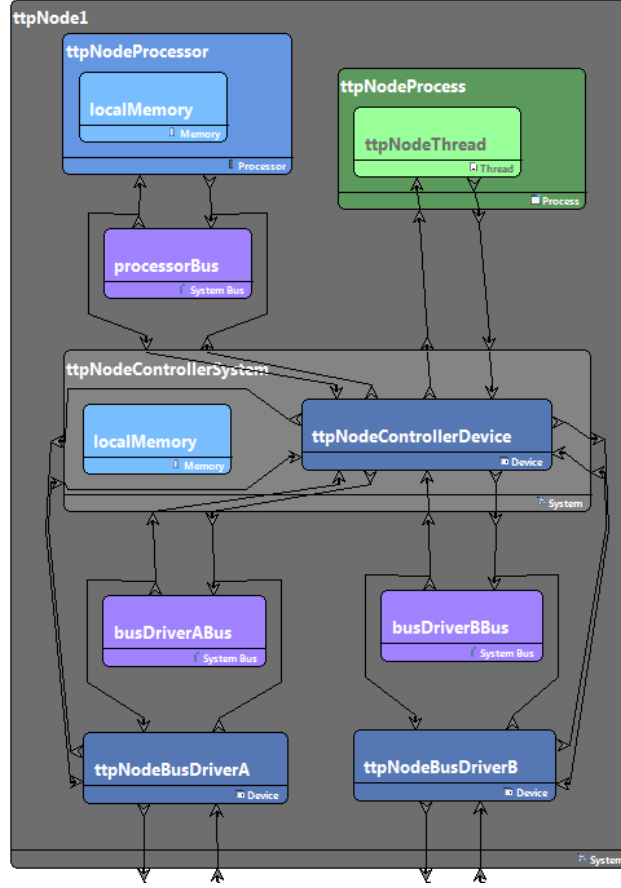


Figure 12. Node Implementation Model

Data dependencies between the `ttpNodeThread` and the `ttpNodeControllerDevice` are modeled using AADL data connections. The hardware connection between the `ttpNodeProcessor` and the `ttpNodeControllerSystem` is modeled as a bus access connection.

8.2.1 Modeling Buses

Similar to SAFEbus, we opted to capture the dual-channel TTP bus as two AADL devices, representing the two channels. The factors that lead to this decision are:

- To bind data flows to the buses, one must specify dependencies between the host applications and, thus, the `ttpNodes`.
- In the TTP model shown in Figure 11, data is traveling through both channels simultaneously; however, the `ttpControllerSystem` merges the two flows into a single flow. AADL does not provide mechanisms to a replicated data flow, so the only option is to introduce independent flows. This approach does not capture the intent behind replication.
- Devices can source errors in the EDICT tool suite, but buses cannot.

8.2.2 TTP Hub Model

Figure 10 shows the TTP hub model. The hub model replicates the TTP-shared bus design shown in Figure 11 and introduces hubs in place of the dual channels. The hubs act as independent guardians, performing Slightly out of Specification (SoS) fault-masking for both value and temporal asymmetric errors. When synchronized, the hubs also enforce TDMA slot access order, and ensure that media will always be available during protocol startup.

The hubs communicate with each other through intra-lane channels. Hubs perform protocol-level services such as prioritizing traffic arriving from different sources, but they are not switches and cannot shut down traffic between the two halves of the hub. The hub's main purpose is to overcome the single point of failure in the `ttpNodeControllerSystem` in the regular TTP shared bus model.

8.3 Modeling Error Propagation in TTP Using the AADL Error Annex

We selected TTP for modeling because of its interesting membership implementation, which is sensitive to Byzantine and SoS faults. A second reason for selecting TTP is that the protocol flow depends on software life-sign strobing, so the protocol fault tolerance is linked to correct software execution. This software protocol interaction is another area of model dependency exploration.

The TTP model starts the processing subsystem, `TtpNodeProcessor`. Since the focus of the modeling is the network dependability, detailed modeling of the application software has not been performed at this stage. Instead, this initial model focuses on the software interaction with the protocol hardware and the potential propagation of protocol faults.

Using basic fault events from the naming convention, the `TtpNodeProcessor` may source temporal and value errors that cannot be detected with inline checks. In these cases, it can exhibit crashing or babbling behaviors. Since the processing system is connected to only one client, we assume symmetric error manifestations. For completeness, it may be argued that faulty processing hardware could source an asymmetric error in the form of a stuck at one half data value that may propagate through low layers, as postulated by [6]; however, we removed this detail to keep the model simple.

In addition to the low-level errors, the implementation of the TTP protocol is also dependent on, and vulnerable to, software actions. Hence, additional semantic error propagations and the associated error states and events are also added to the `TtpNodeProcessor` model. These states and events are denoted with the `sp_m_sw` prefix, indicating that they are high-level errors sourced from the software and/or processor context. The errors are summarized below:

Sp_m_sw_bad_config representing bad configuration for the controller.

Sp_m_sw_reset representing a SW reset.

Sp_m_sw_no_life_sign representing missing life-sign.

sp_m_sw_nc representing noncongruency-different messages on different channels.

The model assumes that both permanent and transient faults can induce any one of these failure states. Hence, separate error events are included for transient and permanent arrival rates. The `TtpNodeProcessor` model also includes error states that can be induced by the underlying communications system:

m.ttp_no_sync representing lost TTP synchronization.

m.ttp_dropped_data representing dropped TTP data.

m.ttp_re_sync representing a TTP re-sync event.

These states are entered following error propagations sourced from the underlying TtpCommunications. Since the TtpNodeProcessor is unable to mitigate such events, no input guards are used, and the error propagations are modeled by simple state transitions within the TtpNodeProcessor core model. Guard events are preferable for linking the software and error models.

The next component that we model is the TtpNodeControllerSystem. This component represents the TTP communications controller IC. It executes the protocol itself and presents the greatest challenge to the modeler.

The validity of the current modeling approach is undetermined at present, since as the model has been populated with detail, it has grown to mirror a low-level abstraction of the TTP protocol itself because the protocol modes of TTP determine how faults are tolerated. For example, a protocol semantic violation at startup can force an entire cluster to fail to integrate. However, once the protocol has reached a synchronous state, with a consistent *c-state* distributed, the shared *c-state* can be used to qualify and reject erroneous protocol semantic frames. Similarly, if both buses are busy (occupied by babbling), the protocol and software will fail to commence synchronous operation.

Once running, synchronous error events from the host and the bus may have different influences. This *c-state* agreement also makes the protocol vulnerable to Byzantine-induced cliques. If a temporal, arbitrary error occurs on both bus channels or such an event occurs on a single channel while the other channel incurs an omission or detected error, the temporal error can induce cliques.

This state is captured in the TtpNodeControllerSystem by the guard events reacting to value-asymmetric and/or temporal asymmetric input propagations. Similarly, the bus impact on the lower-level state transitions is captured with separate guard events. Host-induced failures are also mitigated by guards. Since we assume that the controller is configured to autonomously replicate data, the propagation of software-induced, non-congruent errors (*p_m_sw_nc*) ceases at the TtpNodeControllerSystem. Similarly, since the TtpNode ControllerSystem operates on an autonomous schedule, babbling software faults are contained by the controller.

8.3.1 Driver Modeling

The Simple Driver model represents the bus driver components and associated circuitry. The model is split into separate ingress and egress subcomponents by reducing the model complexity through sourcing input and output error events concurrently with separate error state machines. Similar to other components, the driver model first enumerates the potential error states.

A shorted driver maps to the babbling (*p_b*), since this will result in a denial of bus service for other bus members. An open driver is represented by the *p_o* error.

Since the driver has no notion of time, temporal errors are not assumed. For this model, we also assume that driver-induced value errors will be detectable using the inline error codes; hence, only the detectable errors *p_vsd* are specified. Since the driver connects to a common bus, it is possible for the driver to source an asymmetric value error to the bus. For example, consider a weak driver scenario; some nodes that are close to the driver may receive the data correctly, while nodes that are farther away or impeded by erroneous reflections may not. So a value arbitrary error event *p_va* is sourced from the driver model.

To reduce model driver complexity, only permanent driver errors are assumed for this model. A further implication is that all failure modes occur with an equivalent probability related to the driver permanent failure rate, which is approximated to be 10^{-7} errors per hour.

The driver is a simple component and unable to mask error events. The model therefore maps higher-level error propagations to allow them to be passed through. Note that these pass-through events include temporal errors and the higher-level *m.ttp* errors that relate to TTP protocol-specific

errors.

8.3.2 Channel Modeling

The Channel Model represents the TTP communication channel that relates to the wires and connectors. The error states contributed by the channel are assumed to comprise:

Sp_o permanent omission errors due to a broken cable.

Si_vsd temporary bit flips manifesting symmetrically on the channels due to induced noise. Note that the current model assumes these bit flips are detectable by the inline coverage, hence the suffix **d** is used.

Si_va transient bit flips that manifest asymmetrically on the channel.

p_va permanent faults that divide the channel, to yield asymmetric error value manifestations. This is analogous to a missing bus termination.

The permanent failure rate is assumed to be 10^{-7} failures per hour. The transient failures are assumed to be less frequent as they are proportional to the bus bit error rate; they are assigned a value of 10^{-9} errors per hour. Similar to the driver model, the channel model also declares **in** and **out** error propagations for errors that are not sourced by the channel itself, but are contributed by the higher level driver and TTP protocol components.

8.3.3 TTP_Hub Modeling

The TTP acts as a central guardian for the TTP bus. This is an interesting example from a modeling perspective, since it provides fault-masking properties to the hosted TTP controllers, while itself being dependent on the TTP controllers for operation. For synchronization, the hub uses out-of-band protocol signals such as **m_ttp_action_time**. The first elaboration to the initial TTP model is the addition of the **m_ttp_action_time** signals. Similar to the data signals, these are elaborated with the applicable error modes. Since this is a discrete pulse value error, the error model of the signal is abstracted to erroneous babbling and omission. In addition, since the hub is mode-dependent for fault-free action times, another **m_ttp_m_action_time_err_free** is added. The subsequent event is sourced when the controller is running and fault-free. The full set of action time signals is as follows:

m_ttp_m_action_time_err_free representing action time error free.

m_ttp_m_action_b representing action time babbler.

m_ttp_m_action_o representing action time omission.

At the abstract level, the hub has three states: **unsynchronized**, **synchronized_lost**, and **fully_synchronized**. As with the protocol states, the degree of fault containment performed by the hub is determined by its state; the hub states are presented in the model.

The hub starts in the **unsynchronized** state and will return to that state if two or more erroneous (babbling) action times occur. To reach the **synchronized_lost** state, the hub requires two error-free action times and can tolerate one erroneous action time (babbling). To reach the **fully_synchronized** state, the hub must be in the **synchronized_lost** state and at least two correct nodes must be transmitting data. These conditions are captured in the **In** guard event transitions.

9 Findings and Discussion

This section details the challenges and observations at the end of Year 1. Many of these observations have been addressed as version 2 of the error annex has been developed.

9.1 Benefits and Overheads of the Systematic Fault Taxonomy

As with most model development, a key benefit is the ability to explore the modeled domain. The ability to capture the rationale of design and the assumptions that underpin the model is also important. We found the application of the simple fault taxonomy and naming convention of Section 4.1 to be very beneficial and effective. Exploring the error modes at multiple architectural layers allowed a more systematic *examination of conscience*, making the modeler reconsider the potential failure contributions at each layer.

An interesting side effect of this naming convention is that such a semi-mechanical examination “checklist” yielded a potential error model state explosion as the taxonomy of error failure modes was applied to components that we had thought simple. For example, the modeling of a driver required decomposition into smaller subcomponents to facilitate efficient modeling of concurrent failure manifestations. Attempting to map all ingress and egress error manifestations to a single state machine rapidly became intractable, and a hierarchical decomposition of the driver was required to separate potential concurrent error contributions. For example, a single integrated-circuit quad driver yielded an error model with eight internal error state machines as separate ingress and egress error manifestations were captured. The totality of these eight state machines was much less complex than a single 2^8 input state machine. From our experiences with the driver components, we feel that a generalized method to guide hierarchical decomposition may be beneficial (and potentially critical) if resulting models will remain tractable for analysis.

A second observation about the application of the fault taxonomy is the relatively high syntactic overhead required by version 1 of the Error Annex. The current model requires the declaration of dedicated states, transition and error events for each model behavior, and requires these to be repeated for transient and permanent faults.

The improved Error Type system in the updated version of the Error Annex² appears to address the aforementioned issues. It supplies the structured systematic fault framework and improved syntactic efficiency to aid its application.

9.2 Role of Multiple Layers of Abstraction

To improve error model reuse, we believe that a better layering methodology needs to be developed. We feel that a weakness of the AADL modeling approach is that a driver model must have knowledge of the upper protocol layers. This is illustrated in the TTPs modeling, where protocol-centric failures (i.e., those that were a function of semantic content or timing) required declarations and pass-through mappings within the driver component for protocol error propagation, although an actual driver would have no knowledge of protocol data or time semantics. From a reuse perspective, such mappings introduce semantic layer pollution that precludes component reuse. In the ideal case, a layering hierarchy should be developed to allow greater abstraction and pass-through of higher-layer error events. This would allow a driver model to remain agnostic to specific system $\leftrightarrow$ target instantiations.

One difficulty in developing models without an available execution and analysis environment is completeness. AADLs states that an error specification is erroneous if all input propagation

²In draft at the time of writing.

events are not captured within a guard. Although layering the models more effectively may help by allowing events from different layers to pass through states without explicitly specifying them, it may also complicate the assessment of completeness.

9.3 Completeness of Modeling and Analysis

The question of completeness is further compounded by AADL’s strict ordering of guard actions. In AADL, the order of guard conditions is important, with the first matching guard taking precedence over others below it in lexical order. Although we welcome the rigors of the possible specification, we also believe that this is an area where formal model translation, simulation, and analysis (model checking) will be greatly beneficial to the modeler, ensuring that the intended behavior is what the modeler anticipated.

Similarly, an issue that is already under discussion within the AADL working committee is the ability of a component to query the internal state of a component it is connected to. Using such coupling, it is possible to completely circumvent the AADL Error Annex error propagation mechanism and to code error transitions from coupled state knowledge.

9.4 Obtaining Probabilities

Relative to error probabilities, we have two findings. We found that determination of the probabilities for the esoteric error manifestations was non-trivial, guided more by art than science. In an initial system model (where detailed reliability models and evidence are not available), complex failure modes are often estimated by simple rules of thumb; for example, I expect 1% of my permanent failures to result in babbling.

Currently, such assumptions can be modeled by adding intermediate states to the error model. However, we feel that the ability to express an event occurrence as a function of another event occurrence may be beneficial. For example, using something like:

$$occurrence_{fail_X} = occurrence_{fail_Y} * 0.01$$

which means that the probability of X occurring is 1% of the probability of another SoS error event (Y). In the early states of model development, this may not require all states to sum up to one, but we need to conduct more informal explorations to test the sensitivity to such assumptions.

A similar concept is required for hierarchical composition. By decomposing the state into separate automata, we do not want to infer that the states manifest independently. Instead, we want the probability numbers and distributions to express the failure rate of the hierarchical concurrent child automata. To express such issues in a probabilistic reasoning framework, methods must be developed that equalize probabilities as “weights,” instead of treating them as hard numbers.

9.5 Composition of Heterogeneous Models of Computation

Our work so far has been performed largely bottom-up, focusing on communication connectivity and protocol layers. We feel that a similar methodology would be beneficial if applied top-down, where application and software developers also declare the fault model for the expected communication exchanges using a similar taxonomy. Formalizing the expectations of each layer may then provide for greater application and platform reuse and, in the longer term, automate consistency checking of application requirements with the underlying platform and communication layer guarantees.

A significant challenge identified during our modeling of error propagation and mitigation is the need to compose multiple, potentially heterogeneous MoCs to express the behavior of both the analyzed system and error propagation. The current AADL Error Annex relies on a probabilistic automata context, whereas AADL itself is defined using data-flow-like semantics. The composition

of such models must be captured for formal analysis of error propagation in BRAIN. Moreover, the behavior of the BRAIN nodes themselves must also be captured, potentially through the AADL Behavioral Annex, Finite State Machines (FSMs) or other formal languages.

Furthermore, it may become practically impossible to capture different aspects and multiple levels of abstraction in the same formal model. Reusing verification results from the formal verification of protocol functionality may help “guide” the error propagation analysis. To explore a model of computation suitable for fault-tolerance analysis, the Real-time Availability Integrity Language (RAIL) was evaluated using the BRAIN as a case-study. This work is detailed in Appendix A.

9.6 Experiments Using Integrated Behavior and Probabilistic Models

An associated first-year deliverable from our research is an investigation of PRISM model checker. This study evaluates two approaches to modeling the reliability of the SPIDER fault-tolerant broadcast protocol using the PRISM model checker. Both approaches rely on continuous-time Markov chains, a constraint of the PRISM tool.

The main result from this experiment is that PRISM is perfectly adequate as a reliability analysis tool. The PRISM specification language is expressive enough and enables easy modeling, and the model checker performance is also good. However, since the PRISM tool cannot encode or analyze non-Markovian models, there are limits to this general applicability.

In a second experiment, the application of PRISM to the analysis of a more detailed behavioral model of the SPIDER protocol is conducted. This is applicable to the discussion of the previous section and the desire for an integrated model of computation. The PRISM model developed for this purpose includes fault occurrence and the faulty component behavior that are modeled using non-deterministic assignments. This work concludes that, although it is possible to implement and analyze a fault-tolerant protocol using the presented techniques, the PRISM tool has its limits. Currently, it is uncertain whether the analysis of these integrated models will scale to more complex protocols or systems.

10 Concluding Remarks

The difficulty in expressing protocol-centric failure behavior indicates that if the long-term goal is to use AADL models as key repositories for generating system dependability attributes, more work needs to be done. The modeling of fault-tolerant protocols and systems within AADL introduced several challenges, principally because this level of system behavior is often abstracted below the AADL platform level. Consequently, the initial version of AADL incorporated simple bus abstractions that precluded active behavior³.

The *virtual bus* abstraction, introduced in version 2 of the AADL modeling language, is a great improvement in this regard. The *virtual bus* allows more elaborate behavior to be assumed within the virtual bus bindings. The virtual bus itself may abstract complex lower-level behavior that could be implemented by a combination of lower-level protocols, buses (and/or additional virtual buses) and as required lower-level systems. One nice feature of the bus abstraction is that it enables properties associated with the bus to be associated with all data flows bound to the bus. This allows us to describe the properties of communication at a meta-level and eliminates the need to annotate flows directly.

For example, service properties such as data consistency guarantees can be bound. Alternatively, a liveness property can also be associated with the bus to bound startup and reintegration temporal performance. These properties can then be used as invariants that can be checked against low levels of protocol refinement and implementation detail. For system services common to fault-tolerant systems such as synchronization and group membership, we believe that an abstraction mechanism similar to the *virtual bus* is needed for these important system features.⁴

A major difficulty encountered during the AADL modeling work was interdependency between the Error Annex model and the core system behavioral model. To produce a faithful, high-fidelity representation within the error modeling domain, it was necessary to replicate almost every detail of system behavior. Through the work performed to date, we believe this is the wrong direction, and what is required is an improved semantic linkage between the error modeling domain and the system behavioral domain. A formal semantic model that enables the integration of the different annexes is not yet complete, and although each annex is itself driving to improve its own formal representation, the cross-domain/annex linkages are at a very early stage. Without such linkages, evidence produced by processing the annotations of one annex in isolation may be incomplete, or worse yet incorrect. For example, the assumptions underpinning an FMEA or fault-tree analysis may be inconsistent with respect to the real system behavior. Reproducing the information within both domains leads to further possibility of inconsistencies and arduous non-value-added modeling overhead.

To this end, developing the cross-annex integration framework between system behavior and error models has been selected as a major area of research for the second phase of this research effort.

³The bus component of AADL is not permitted to source events.

⁴In practice, in real systems, these properties and services guarantees associated with the bus would not be absolute but instead assured; that is, claimed with a degree of probability.

11 Acronyms and Initialisms

AADL Architecture Analysis & Design Language
BIU Bus Interface Unit
BRAIN Braided Ring Availability Integrity Network
BTL Backplane Transceiver Logic
BDD Binary Decision Diagram
CPU Central Processing Unit
CRC Cyclic Redundancy Check
DES Discrete Event Simulation
DSML Domain-specific Modeling Language
EDICT Error Detection Isolation Containment Types
FADEC Full Authority Digital Engine Control
FSM Finite State Machine
GUI Graphical User Interface
LAN Local Area Network
LRM Line Replaceable Module
MAC Medium Access Control
MoC Model of Computation
nMR n-Modular Redundant
OSATE Open Source AADL Tool Environment
PE Processing Element
RAIL Real-time Availability Integrity Language
RMU Redundancy Management Unit
ROBUS Reliable Optical Bus
SAL Symbolic Analysis Laboratory
SCP Self-checking Pair
SoS Slightly out of Specification
SPIDER Scalable Processor-Independent Design for Enhanced Reliability
TDMA Time Division Multiple Access
TMR Triple Modular Redundancy
TTP Time-triggered Protocol
XML Extensible Markup Language

References

1. AS-2 Embedded Computing Systems Committee SAE: Architecture Analysis & Design Language (AADL). SAE Standards N° AS5506A, January 2009.
2. AS-2 Embedded Computing Systems Committee SAE: Architecture Analysis and Design Language (AADL) Annex Volume 1. SAE Standards N° AS5506/1, June 2006.
3. Joshi, A.; Binns, P.; and Vestal, S.: Automatic Generation of Fault Trees from AADL Models. *Proc. Aerospace Software Engineering Workshop*, 2008.
4. Rugina, A.-E.; Kanoun, K.; and Kaâniche, M.: The ADAPT Tool: From AADL Architectural Models to Stochastic Petri Nets through Model Transformation. *CoRR*, vol. abs/0809.4108, 2008.
5. Hecht, M.; Lam, A.; Howes, R.; Vogl, C.; Lake, S.; and City, U.: Automated Generation of Failure Modes and Effects Analyses from AADL Architectural and Error Models. *Proc. 2010 Systems and Software Development Conference*, 2010.
6. Driscoll, K.; Hall, B.; Paulitsch, M.; Zumsteg, P.; and Sivencrona, H.: The Real Byzantine Generals. *In Proceedings of the Digital Avionics Systems Conference*, 2004, pp. 61–71.
7. Paulitsch, M.; Morris, J.; Hall, B.; Driscoll, K.; Latronico, E.; and Koopman, P.: Coverage and the Use of Cyclic Redundancy Codes in Ultra-Dependable Systems. *In Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2005.
8. Bauer, G.; Bilic, K.; Driscoll, K.; Salloum, C. E.; Eles, P.; Elmenreich, W.; Goller, A.; Hall, B.; Kammerer, R.; Kantz, H.; Kopetz, H.; Obermaisser, R.; Paulitsch, M.; Pop, P.; Pop, T.; Scherrer, C.; Schmidt, E.; and Steiner, W.: *Time-Triggered Communication*. Embedded Systems Series, CRC Press, Taylor & Francis Group, 2011.
9. Torres-Pomales, W.; Malekpour, M. R.; and Miner, P. S.: NASA/TM-2005-213540 ROBUS-2: A Fault-Tolerant Broadcast Communication System. 2005.
10. Kopetz, H.; and Bauer, G.: The Time-Triggered Architecture. *Proceedings of the IEEE*, vol. 91, no. 1, 2003.
11. TTA Project: The EU-funded OMI project 23396 TTA (Time-Triggered Architecture) aimed at the implementation of a time-triggered computer architecture (TTA) for fault-tolerant distributed real-time systems. <http://www.vmars.tuwienvmars.tuwien.ac.at/projects/tta/index.html>, 1998.
12. Safety Related Fault Tolerant Systems in Vehicles - X-by-wire (Project Ref. BRPR950032, Brite-EuRam III). <http://www.vmars.tuwien.ac.at/projects/xbywire/index.html>, 1998.
13. Fault Injection for TTA (FIT). Project Ref. IST-1999-10748, 2000.
14. Full Authority Engine Control Systems: Honeywell's TTT-based modular aerospace control. <http://www.tttech.com/fileadmin/content/pdf/TTTech-Honeywell-Casestudy-MAC.pdf>.

Appendix A

Real-time Availability Integrity Language (RAIL)

A.1 Introduction

This appendix presents a semantic domain for evaluating fault-tolerant systems. Our goal is to provide an analysis framework that can formally express both low-integrity and high-integrity data communication. In particular, we capture the following concepts for modeling:

Tokens: We rely on the notion of tokens to model message exchange in distributed systems. Tokens may have several properties associated with them that we use to characterize the message flow.

Token colors: We use the concept of token colors to capture concepts such as data integrity, voting, and fault modeling.

Token priorities: We introduce token priorities in the model to distinguish between the categories of data criticality. Such priorities allow us to model traffic-shaping algorithms and to express that high-criticality data has preference over noncritical data.

We refer to the proposed MoC as Real-time Availability Integrity Language (RAIL). We describe fault-tolerance-specific extensions to RAIL in Section A.3. We demonstrate how token colors can be applied to express low- and high-integrity data. We demonstrate the feasibility of the approach on a simple example based on a braided ring topology.

A.2 Background

The purpose of this work is to capture three key design aspects of distributed fault-tolerant systems: *availability*, *integrity*, and real-time properties in both synchronous and asynchronous systems.

Availability: Availability can be represented as a logical OR gate; if data is present on any of the OR gate’s inputs, the data is propagated through the gate. In fault-tolerant systems, the use of replication in combination with OR gates results in high availability; some data will propagate through the OR gate unless all inputs are silent or faulty.

Integrity: Integrity can be represented as a logical AND gate; data on all inputs must match for data to propagate through the AND gate. The AND gate essentially performs a *comparison* of input values in order to confirm whether input data is consistent. A generalization of this idea is to use *voters* that vote to determine what is the proper value in case not all inputs match.

In fault-tolerant distributed systems, both high availability and high integrity are desirable. In this work, we propose a semantic domain based on discrete event systems that can model the dynamic relationship between high integrity and high availability in a large class of distributed systems based on mesh topologies.

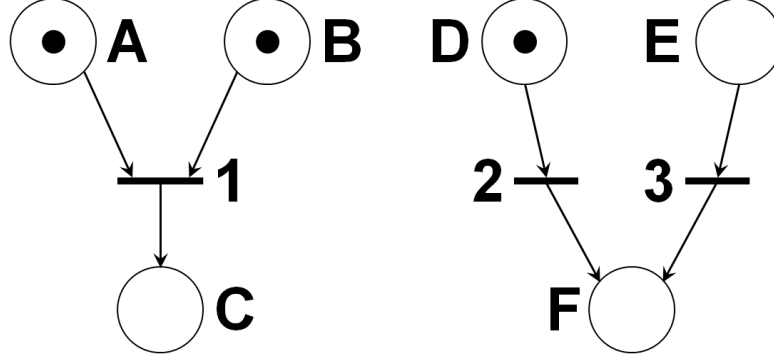


Figure A1. Petri-net Model of AND and OR Behavior

A.2.1 Discrete Event Systems

A Discrete Event Simulation (DES) system can be expressed as the tuple $M = \{S, s_0, E, T, \Sigma, \delta\}$:

- S is the set of states,
- $s_0 \in S$ is initial state,
- E is the set of events,
- $T : S \times E \times S$ is the set of transitions,
- Σ is a finite alphabet of symbols called event labels,
- The labeling function $\delta : E \rightarrow \Sigma$ specifies event labels for events.

In DES systems, transitions depend only on the current state and the event label. There is no explicit notion of time, although a partial ordering is implied by the order of events and transitions; however, the formalism can be extended in multiple ways by extending the event labels. For example, the event label can be used to denote (possibly real-valued) timestamps, or probabilities. In the following subsections, we explore two popular MoCs commonly applied to the modeling of DES systems, and describe some extensions aimed at expressing dynamic high-availability and high-integrity systems.

A.2.2 Petri-nets

Petri-nets are a popular MoC for modeling concurrent discrete event systems. They are a natural fit to model event-based communication in distributed systems. Petri-nets provide a way to model both availability OR and integrity AND behavior. To express availability, a place may consume tokens from multiple transitions, thus expressing independence between various data flows. Moreover, Petri-nets can also model integrity AND behavior through a transition that consumes tokens from potentially multiple places.

The left side of Figure A1 demonstrates how Petri-nets are able to capture integrity AND behavior; tokens on places A and B must be present to enable transition 1 for firing. When firing, transition 1 consumes both tokens from places A and B and produces a token in place C.

The right side of Figure A1 shows how Petri-nets can specify availability OR behavior. Place D is connected to place F through transition 2, and place E is connected to place F through transition

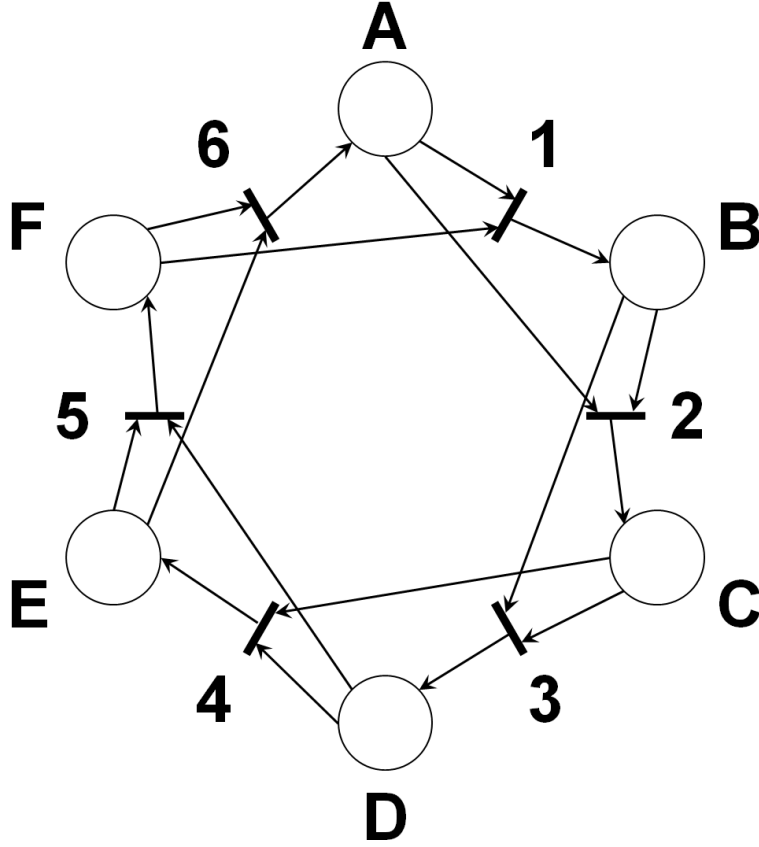


Figure A2. Petri-net Model of High-integrity Ring Network

3. Transition 2 and transition 3 get enabled and fire independent of each other. As a result, if a token is only present in place D, transition 2 is enabled and ready to fire.

The behavior of the two examples above is very distinct; the AND example on the left is capable of modeling simple voting mechanisms by expressing that data from multiple sources must be processed together, whereas the OR example on the right can express that data can reach place F even in the case of failure, where place E is not generating any tokens.

A.2.3 Applying Petri-nets for the Modeling of Ring Networks

Figure A2 shows the Petri-net model for a high-integrity, ring-topology, distributed system. The model consists of six places and six transitions. The transitions model high-integrity message passing along the ring; each transition having two places as their sources.

For example, transition 1 is enabled if both place A and place F contain at least one token each. When transition 1 fires, it consumes two tokens; one from place A and one from place F; and produces one output token in place B. This mechanism can abstractly capture a voter with two inputs; if the two inputs match, the data is treated as high-integrity, otherwise the data is low-integrity and not trusted.

Token Propagation Along the Ring Transition 2 shown in Figure A2 is enabled and ready to fire when both place A and place B contain a token. When firing, transition 2 consumes the token from place A and place B and produces a token in place C.

After this step, the model deadlocks. There is only one token present in place C, therefore neither transition 3 or transition 4 is enabled for firing. Thus, the model shown in Figure A2 does not properly capture the design intent of modeling data propagation along the ring and demonstrates the difficulty of applying Petri-nets directly for the modeling of complex distributed fault-tolerant systems.

A.2.4 Finite State Machines

A Finite State Machine (FSM) is an alternative to the Model of Computation (MoC) for modeling DES systems. The FSM concept is based on the notion of states and state transition. Basic FSMs are commonly extended with transition guards and synchronized transitions in order to express multiple, concurrently executing automata. Such extensions build on a network of FSMs, that can exchange events in either a broadcast or multicast fashion. Such extensions are commonly applied to practical model checker tools, such as SRI’s Symbolic Analysis Laboratory(SAL) and NuSMV.

A.3 Modeling Fault-Tolerant Communication in Distributed Systems

This section describes the RAIL Domain-specific Modeling Language (DSML). RAIL is a language for modeling and analysis of high-integrity distributed systems. We capture the notion of availability and integrity and provide a way to verify real-time constraints in a large class of mesh-based distributed systems.

A.3.1 Applying RAIL for the Analysis of Braided Ring Topologies

A braided ring is one of the simplest mesh topologies. We chose braided ring topology for this study as we hope the results can be generalized to more complex mesh architectures. In a braided ring, each node is connected not only to its immediate neighbor, but to its second neighbors as well.

In this study, our goal is to create a semantic domain that can express mixed-integrity message passing on braided ring topologies. The approach should be able to provide an abstract representation of voted-integrity architectures as well.

For this study, we allow simplex nodes to act as senders/receivers. We also want to capture the voting mechanism used to validate links, with the possibility of generalizing the concept to n-Modular Redundant (nMR) voting architectures.

The notion of integrity used in RAIL is based on independence; nodes confirm integrity by comparing messages received on independent paths. If a message is received on a single path only, it is treated as low-integrity until it can be confirmed through an independent path.

Figure A3 demonstrates how a braided ring topology can be captured using RAIL. Nodes are denoted by circles (A - F). Arrows represent model token propagation and are referred to as *connections*. Each node is modeled using two token queues, corresponding to event passing in a certain direction along the ring. Connections resemble the braided ring topology by connecting each node to its immediate and immediate next neighbors.

A.3.2 RAIL Execution Semantics

In this section, we demonstrate the execution semantics of RAIL through a simple example based on the braided ring topology. We build on token colors to distinguish between different types of tokens. We also use the term “hot” to refer to a token that is ready to fire. While all tokens could

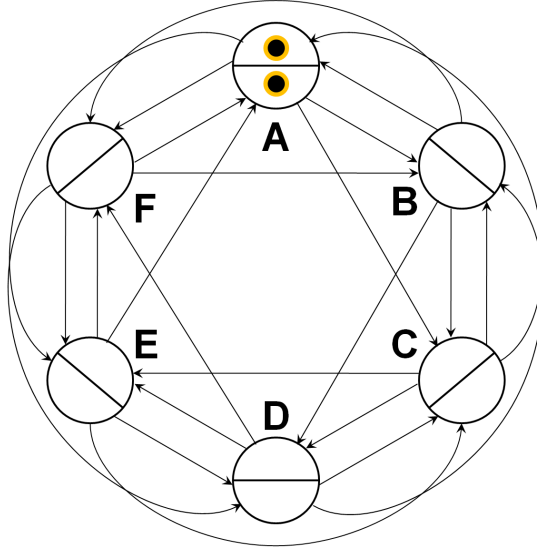


Figure A3. RAIL Model of Braided Ring Topology

be mapped to an arbitrary color, we found the “hot” designation easier to follow. In its current form, RAIL builds on the following token descriptors:

Black: represents a high-integrity token that was confirmed on two independent paths.

Black_hot: represents high-integrity token. The node containing this token is ready to fire.

Gray: represents a low-integrity token. This token was received from a single source only. If the node receives another gray token, then it becomes a black (high-integrity) token.

Gray_hot: represents a low-integrity token. The node containing this token is ready to fire.

Blue_hot: represents a high-integrity token. The node containing this token is ready to fire; however, this token will propagate backwards in the opposite direction on the ring. This behavior models a guardian and is explained in more detail in Step 3 below.

Yellow: An existing high-integrity token receives another low-integrity token— a short-hand notation to describe when both black and gray tokens are present within a node. We introduced this color to simplify the figures and the SAL proof. This token models the case when a message successfully propagated through the ring and is now received by the original sender from one direction.

Green: An existing high-integrity token receives another high-integrity token. This is a short-hand notation describing when two black tokens are present within a node. We introduced this color to simplify the figures and the SAL proof. This models the case when messages successfully propagated through both directions in the ring.

Figure A4 demonstrates what we refer to as a “round” of message propagation through the ring. The execution sequence is from left to right, then from top to bottom.

Step 1: Node A contains a **black_hot** token ready for propagation in both directions. This models that node A in the topology is ready to start sending messages on the ring.

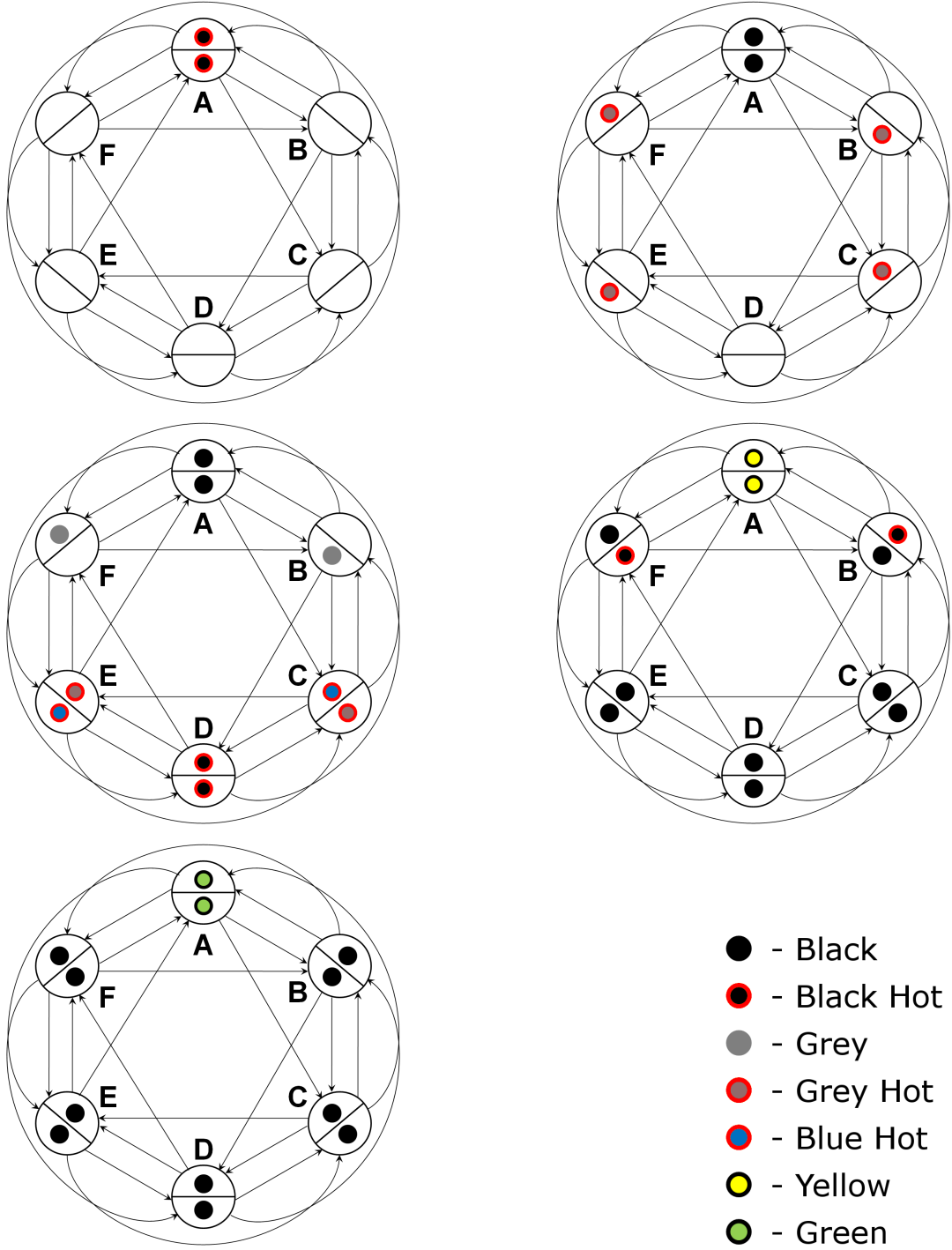


Figure A4. Demonstrating RAIL Execution Semantics on Braided Ring Topology

Step 2: Connections originating in node A fire. Since there was only a simplex sender, the message received is not confirmed on either link. Thus, low-integrity `gray_hot` tokens are created in nodes B, C, F, and E. The `black_hot` token in node A cools down after firing, and becomes a `black` token.

Note the directional separation; in nodes B and C the tokens are present in the locations

corresponding to clockwise data propagation, whereas in nodes F and E data is propagating counter-clockwise. All new tokens are hot and enabled for firing.

Each token is kept in its corresponding location for the duration of the round. Once the round is complete, the whole ring is reset, and all tokens are removed. This approach models store & forward behavior and plays a role in high-integrity data reconstitution as explained in Section A.3.4.

Step 3: Node D now contains `black_hot` tokens for both directions because D has received `gray` tokens from both B and C in the clockwise direction, and E and F from the counterclockwise direction. Both directions are thus independently confirmed on two independent paths.

The tokens in nodes C and E are now also high-integrity, as C receives `gray` tokens from both nodes A and B. Likewise, node E receives `gray` tokens from both nodes A and F.

The high-integrity tokens are `blue_hot` in both nodes C and E. Blue tokens model guardian behavior. In this case, both nodes are enabled to fire tokens backward. The rationale for this can be seen from the braided ring topology.

Given that we have allowed simplex senders in RAIL, the immediate neighbors of the sending node A cannot receive tokens from independent paths from one direction, unless the token propagates through the whole ring. To alleviate this restriction, the guardian bounces back a token to enable A's neighbors to confirm high-integrity message passing. Thus, the `blue_hot` token in node C will result in a token propagating back to node B and the `blue_hot` token in node E will result in a token propagating back to node F. Nodes C and E also receive `gray_hot` tokens in the other direction through normal token propagation, and so do nodes B and F.

Step 4: Nodes B and F receive the backward propagating token from nodes C and E. Coincidentally, they also receive high-integrity tokens from the other direction. With topologies consisting of more than six nodes, these two steps would not occur simultaneously. The tokens in node A turn yellow, indicating that A has received a token from nodes C and E.

Step 5: The tokens in node A turn green, indicating that A has received additional low-integrity tokens, this time from nodes B and F. The round is now complete, and A has received confirmation that data propagation was successful in both directions. The latter condition does not necessarily hold in all fault scenarios. The ring is then reset, and propagation may commence, potentially from a different source.

A.3.3 Comparison with Petri-nets

RAIL is based on the FSM MoC, but builds on the notion of tokens to characterize message flow. Petri-nets also build on tokens, but are not as expressive as RAIL. A key property of RAIL is the queuing of tokens, that allows the modeling of both reconstitution and priority-based arbitration.

The right side of Figure A5 shows how Petri-nets can specify availability OR behavior. Node D is connected to node F through transition 2, and node E is connected to node F through transition 3. Transition 2 and transition 3 get enabled and fire independent of each other. As a result, if a token is only present in node D, transition 2 is enabled and ready to fire.

RAIL, on the other hand, can capture both low- and high-integrity message propagation on the same path. Thus, it can express both sides of Figure A5. Moreover, Petri-nets do not distinguish between token priorities. In short, the extensions necessary to model RAIL in Petri-nets essentially turn the model into a network of FSMs.

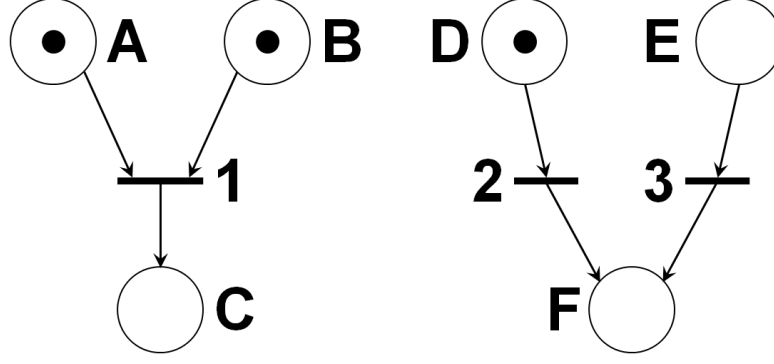


Figure A5. Petri-net Model of AND and OR Behavior

A.3.4 Reconstitution

Reconstitution plays an important role in providing high-integrity message exchange in the presence of multiple faults. In certain circumstances, a node is unable to obtain high-integrity tokens from just one direction in the ring. In these cases, the node may be able to “reconstitute” a high-integrity token from two low-integrity tokens that it has received from different directions among the ring. For example, one low-integrity token may have reached the node using clockwise propagation, whereas the other low-integrity token may have been received in a counterclockwise fashion. Currently, we define reconstitution rules as follows:

```

token' =
  IF (left_token = green) OR (right_token = green) THEN
    green
  ELSIF (left_token = yellow) AND (right_token = yellow) THEN
    green
  ELSIF (left_token = yellow) OR (right_token = yellow) THEN
    yellow
  ELSIF (left_token = black) OR (left_token = black_hot) OR
    (left_token = blue_hot) OR (right_token = black) OR
    (right_token = black_hot) OR (right_token = blue_hot) THEN
    black
  ELSIF (((left_token = gray) OR (left_token = gray_hot)) AND
    ((right_token = gray) OR (right_token = gray_hot))) THEN
    black
  ELSIF (left_token = gray) OR (left_token = gray_hot) OR
    (right_token = gray) OR (right_token = gray_hot) THEN
    gray
  ELSE
    empty
  ENDIF;

```

The next value of the token depends on the value of both `left_token` and `right_token`, expressing the values of tokens received from both directions. The basic idea is that two low-integrity tokens within the same node serve as a high-integrity token. Thus, two `gray` tokens lead to a reconstituted `black` token. Likewise, two `yellow` tokens within a node are equivalent to one `green`

token, as the sender receives confirmation from both directions that the low-integrity token propagation was successful in both directions. This flexibility is essential in providing fault-tolerance in the presence of two faults, where the ring topology can break up in unexpected ways.

A.4 Automated Verification of RAIL Models

This section describes methods that could be applied to the analysis of RAIL models. We apply model checking methods based on the SAL model checker as described below. Alternatively, DESs provide an alternative approach for simulation-based evaluation, as described earlier. Note that simulation-based evaluation can also be exhaustive, and in some cases may provide better scalability than Binary Decision Diagram (BDD)-based FSM model checking.

A.4.1 Formal Modeling of RAIL in SAL

RAIL provides for the automated analysis and verification of fault-tolerant distributed real-time systems. To facilitate verification, we have specified the formal semantics of RAIL by creating a representation of the braided ring topology in SRI’s Symbolic Analysis Laboratory (SAL).

SAL is a model checker tool that operates on an extended FSM formalism. We have found that SAL was expressive enough to capture the semantics of RAIL, and we were able to use the resulting models to prove simple fault-tolerance properties.

The SAL model is a textual FSM representation of the braided ring topology shown in Figure A3. The node with `id` of 1 is the first node to propagate tokens through the ring, according to the execution trace demonstrated in Figure A4. Once the whole round is complete, the ring is reset, and the token propagation is continued with node 2 as the sender. Eventually, all `n` number of nodes get to broadcast tokens on the ring network. Thus, when checking properties using the SAL model, one can evaluate all scenarios, regardless of which node acts as the sender.

Properties we have already checked on the SAL model include:

- Verify correctness of TDMA schedule.
- Verify that all nodes receive high-integrity data using a single fault assumption. For this proof, we considered the following scenarios: *(i)* direct link failure, *(ii)* skip link failure, *(iii)* node failure (fail-stop). In the case of node failure, we only guarantee high-integrity data for non-faulty nodes.
- Show that the sender eventually receives confirmation on the success of propagation (i.e., it contains two **green** or **yellow** tokens). Naturally, this condition does not hold unless connectivity in the ring is still available.

As part of this study, we plan to generalize results to analyze the following conditions:

- Verify that all non-faulty nodes receive high-integrity data in the presence of two faults. We restrict fault assumptions to exclude two simultaneous malicious faults.
- Representing babblers using the token-based approach is cumbersome. A potential approach to identify babblers is to introduce hop count information in the tokens, and use that to identify potential babblers.

A.5 Conclusion

We have presented a method to capture low- and high-integrity data in fault-tolerant distributed systems. We have demonstrated how RAIL can capture data propagation along the braided ring topology. We plan to generalize this approach to arbitrary mesh architectures.

As part of our initial work, we have created SAL models for RAIL and were successful in proving simple fault-tolerance properties.

We are currently assessing the feasibility of extending this formalism to capture additional real-time properties that will facilitate the capture of mixed synchronous and asynchronous systems.

REPORT DOCUMENTATION PAGE					Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY) 01-02-2013		2. REPORT TYPE Contractor Report		3. DATES COVERED (From - To) 10/2010 - 03/2011		
4. TITLE AND SUBTITLE Investigating System Dependability Modeling Using AADL				5a. CONTRACT NUMBER NNL10AB32T		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Hall, Brendan; Driscoll, Kevin R.; Madl, Gabor				5d. PROJECT NUMBER		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER 534723.02.02.07.30		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, Virginia 23681				8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001				10. SPONSOR/MONITOR'S ACRONYM(S) NASA		
				11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/CR-2013-217961		
12. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified - Unlimited Subject Category 62 Availability: NASA CASI (443) 757-5802						
13. SUPPLEMENTARY NOTES Langley Technical Monitor: Paul S. Miner						
14. ABSTRACT This report describes Architecture Analysis & Design Language (AADL) models for a diverse set of fault-tolerant, embedded data networks and describes the methods and tools used to created these models. It also includes error models per the AADL Error Annex. Some networks were modeled using Error Detection Isolation Containment Types (EDICT). This report gives a brief description for each of the networks, a description of its modeling, the model itself, and evaluations of the tools used for creating the models. The methodology includes a naming convention that supports a systematic way to enumerate all of the potential failure modes.						
15. SUBJECT TERMS AADL; Dependability modeling; EDICT; Error modeling; Error propagation						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)	
U	U	U	UU	68	19b. TELEPHONE NUMBER (Include area code) (443) 757-5802	